UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Doble Grado en Matemáticas e Informática

TRABAJO FIN DE GRADO

PARALELIZACION Y OPTIMIZACION DE UN CODIGO DE SIMULACION DE IMÁGENES STM

Rubén Mínguez Rodríguez
Tutor: Pablo Pou Bell
Ponente: Iván González Martínez

Julio del 2015

PARALELIZACION Y OPTIMIZACION DE UN CODIGO DE SIMULACION DE IMÁGENES STM

AUTOR: Rubén Mínguez Rodríguez
TUTOR: Pablo Pou Bell

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio del 2015

Resumen

Para el desarrollo de la nanotecnología, en la que ya estamos inmersos con ejemplos como los procesadores actuales de 14 y 22 nm o las baterías basadas en grafeno es necesario el estudio de la materia y la energía a niveles aún más pequeños para saber con certeza el comportamiento de estos dispositivos y limitar el ruido y las imperfecciones de estos.

Una de las herramientas más utilizadas para ello es el Microscopio de efecto túnel (en inglés: Scanning Tunneling Microscope o STM) con el que se pueden estudiar superficies metálicas, semiconductoras y algunas otras a escala atómica (del entorno de los 0,1 nm).

Pero estudiar la materia a esa escala y en condiciones reales tiene sus problemas, por eso la física cuántica teórica intenta solucionar esos problemas mediante el uso de modelos y simulaciones que calculan de forma teórica y aproximada el resultado del uso de un microscopio de efecto túnel sobre una superficie determinada.

Típicamente para ello se han usado modelos teóricos que simplifican demasiado la simulación para ciertas necesidades, por lo que el departamento de Física de la Materia Condensada de la Universidad Autónoma de Madrid (UAM) optó por desarrollar un modelo propio basado en el formalismo de Green-Keldysh de no equilibrio y en la Teoría del Funcional de la Densidad (DFT).

Gracias a esto y debido a la gran cantidad de cómputo necesario para hacer las simulaciones (del entorno de días e incluso semanas por simulación), en este TFG nos hemos propuesto optimizar y paralelizar la simulación de forma que pueda tardar mucho menos tiempo que actualmente.

Para ello se ha optado por el uso de MPI que es el sistema más eficaz para paralelizar las simulaciones gracias a su alta escalabilidad, gran rendimiento y su facilidad de uso.

Con ello hemos conseguido reducir el tiempo de ejecución de las simulaciones en un orden de magnitud (tardando una décima del tiempo anterior).

Gracias a esto hemos conseguido que simulaciones que antes eran impensables de realizar en la física teórica (debido a la ingente cantidad de tiempo de computación) como algunas que tardaban un mes ahora se obtengan en solo 3 días.

Por tanto hemos conseguido no solo reducir el tiempo de ejecución de las simulaciones sino además permitir a los físicos teóricos realizar simulaciones mucho más complejas que las que realizaban anteriormente.

Palabras clave

MPI, profiling, fortran, optimización, paralelizado, rendimiento.

Abstract

For the development of the nanotechnology, in which we are immerse (for instance, the actual microprocessors of 14 and 22 nm or the new lithium-graphene batteries), a study of material and energy in a much smaller level is required, so we can be sure of these products' performances in real conditions.

One of the most used tool for this purpose is the Scanning Tunneling Microscope (also known as STM). Thanks to it, we can study the quality of things at an atomic scale (about 0.1 nm).

However, in real conditions, studying the matter qualities with a STM have some issues. For this problem, physics make use of theoretical physics to make models and simulations of how an STM would work under ideal conditions.

Since this models are usually simplified too much, the Physics Department of Condensed Matter of the Universidad Autónoma de Madrid (UAM) have developed their own model, based on the non-balance of Green-Keldysh theory and the Density Functional Theory (DFT), in order to obtain better measures.

Due to the huge amount of computation the simulations need (days or even weeks), in this TFG we have developed an optimized and parallelized version of this simulation.

Thanks to it and the use of MPI to parallelize the simulation, we are going cut down the simulation time required.

As a result of this work we have reduced the time by an order of magnitude (a tenth of the time needed before).

Using the new program, we are able to generate simulations that could be near impossible (because they could last about a month and could be interrupted by external issues) in only 3 days.

As a conclusion, physics are now able to generate simulations in way less time and much more complex simulation that they could do before.

Keywords

MPI, profiling, fortran, optimization, parallelize, performance.

Agradecimientos

Agradezco a Pablo Pou Bell (tutor), Iván González Martínez (ponente), Rubén Pérez Pérez y Lucía Rodrigo Insausti por su trabajo, ayuda y entrega para conseguir que este trabajo se realice de la mejor manera posible.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLA

1.Introducción

En este capítulo se empezará explicando el funcionamiento de un microscopio de efecto túnel (STM) y el proceso por el que se simula computacionalmente este. Después se expondrá la motivación, los objetivos y los retos. Para finalizar se explicará la estructura del documento.

1.1. Funcionamiento STM

Un microscopio de efecto túnel (en inglés: Scanning Tunneling Microscope o STM) es un dispositivo capaz de tomar imágenes de una superficie a nivel atómico.

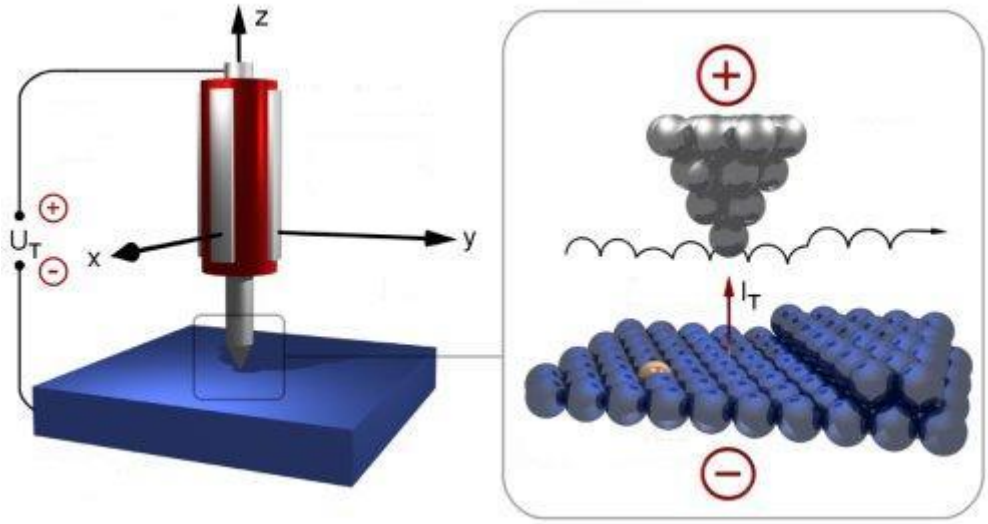

Figura 1: Imagen sobre el funcionamiento de un STM

Como se puede apreciar en la Figura 1 el STM está compuesto principalmente por una punta (normalmente terminada en un átomo) a la que se le aplica un voltaje distinto de la superficie a estudiar. Con ello lo que se consigue es que a distancias muy cortas (de 0.1 nm a 10 nm, es decir a distancia de unos átomos) entre la punta y la superficie, esa diferencia de tensión induce una corriente que se transmite entre la punta y la superficie a través del vacío, aire agua u otros gases mediante el llamado efecto túnel (que solo se produce a estas distancias). Esta corriente entre la punta y la superficie varía de forma exponencial con la distancia entre los dos elementos. Esto es muy importante ya que gracias a que la intensidad cambia muy rápidamente moviendo muy poco la punta, se puede obtener una resolución suficiente para ver los átomos.

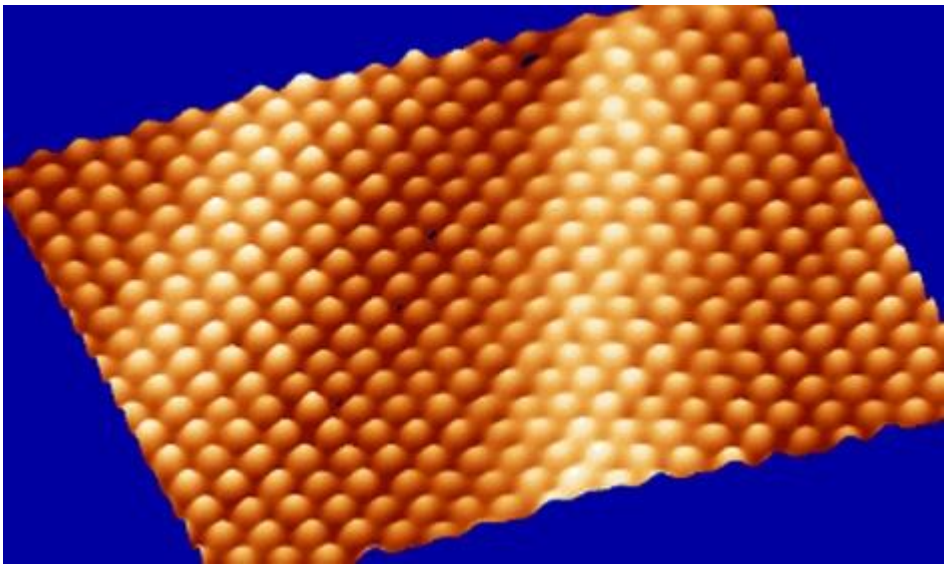

Figura 2: Imagen de una superficie de grafeno

La Figura 2 muestra una imagen STM de una superficie de átomos uniformemente distribuidos. Como se puede apreciar, lo que se mide es la densidad electrónica de la superficie que presenta sus máximos en las posiciones de los átomos.

Con estos microscopios se pueden estudiar materiales metálicos o semiconductores ya que la corriente tiene que fluir entre la punta y el material. Aun así también se puede usar para estudiar ciertos materiales aislantes. Para ello se deposita una fina capa del material a estudiar sobre una superficie metálica que será la que permita que la corriente fluya.

El procedimiento (o pasos) por el cual se crea la imagen con el STM es la siguiente:

1. Se deposita la superficie a estudiar en el STM. La superficie debe ser lo más plana posible y debe estar lo más limpia posible para reducir los errores de las mediciones.
2. Se pone la punta a una distancia del orden de 0.1 nm de la superficie. Este paso es importante ya que si se acerca la punta demasiado esta se puede romper y si está demasiado alejada no se obtendrán mediciones.
3. Usando unos piezoeléctricos la punta es movida en el eje X e Y (y en algunos casos en el Z) obteniéndose una señal (de la corriente) que será de mayor valor donde estuviese cerca de un átomo y menor donde esté lejos de los átomos aunque esto depende también de las propiedades electrónicas locales de estos átomos. Por ejemplo un átomo que fuera “aislante” inducirá una corriente mucho menor que otros átomos “conductores”. Mediante uno o varios barridos sobre la superficie del material se puede obtener una aproximación bastante buena de donde están los átomos y de sus propiedades electrónicas pudiendo generar la imagen de la superficie.

1.2. Simulación teórica

Mediante el uso del STM los físicos pueden obtener imágenes de superficies como las de la Figura 3.

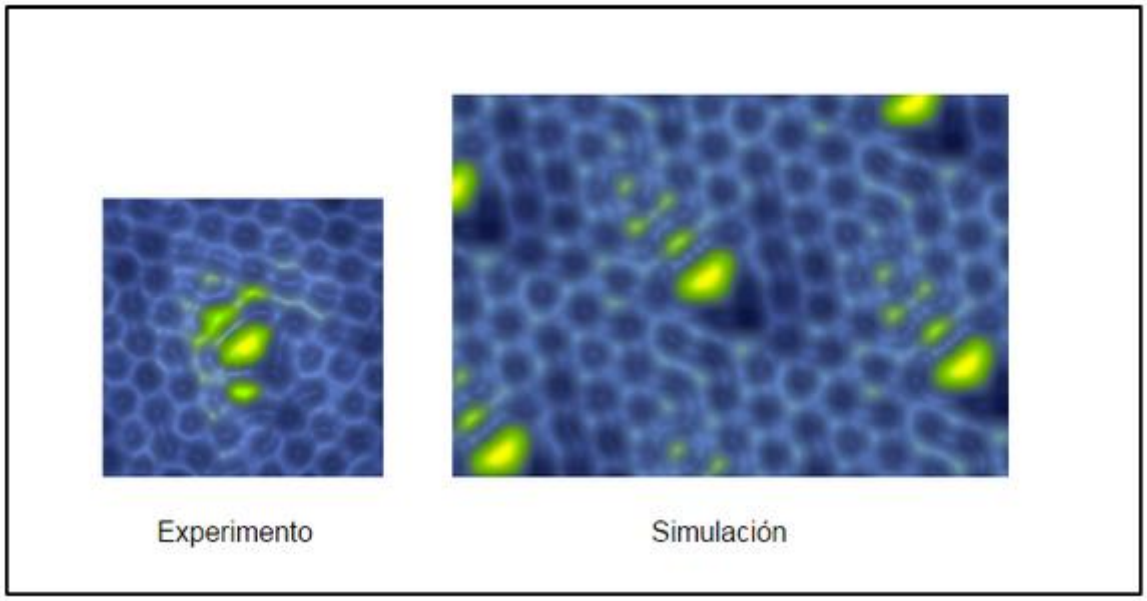

Figura 3: Imagen de un experimento real y su simulación

La primera imagen (experimento) de la Figura 3 muestra el resultado del uso de un STM sobre la superficie de una lámina de grafeno a la que se le ha retirado algunos átomos (vacantes) mediante un bombardeo de protones. Los átomos del grafeno, y los enlaces entre ellos, son las protuberancias azul claro de forma hexagonal. La creación de una vacante de un solo átomo se observa en la imagen STM como una gran protuberancia (de color verde en la imagen).

El problema que les surge a los físicos con esta imagen es que no saben bien que pasa en las zonas cercanas a esos vacíos, hace falta una descripción teórica de estos experimento para obtener una buena descripción de lo que está midiendo el STM y así poder entender mejor las propiedades de los sistema estudiados que en este caso podemos ver en la Figura 3 en la segunda imagen (simulación).

Esta imagen es el resultado de una simulación computacional basada en unos modelos teóricos de la superficie que hemos explicado antes. Estas simulaciones, gracias al gran acuerdo alcanzado con las medidas experimentales, permiten a los físicos entender qué está pasando realmente en el grafeno al crear esas vacantes.

Existen varias aproximaciones teóricas para realizar estas simulaciones de los experimentos STM. Las más sencillas suponen que la punta del STM es solo un punto y la imagen simulada solo tiene en cuenta las características electrónicas de la muestra. Estas aproximaciones aunque funcionan de forma cualitativa en muchos casos, dan resultado incorrectos en muchos otros casos. Simulaciones usando aproximaciones más complejas se han venido realizando con éxito durante muchos años, sin embargo la

complejidad conlleva un aumento de las necesidades computacionales limitando tanto el tamaño como la precisión de los sistemas estudiados.

1.3. Método de simulación propio

Investigadores del departamento de Física Teórica de la Materia Condensada de la Universidad Autónoma de Madrid (UAM) han desarrollado durante años, comenzando desde prácticamente la invención del STM, una metodología de simulación combinando el cálculo de funciones de Green fuera del equilibrio usando el formalismo de Keldysh con cálculos basados en la Teoría del Funcional de la Densidad (DFT) que permite simular con gran precisión cuantitativamente estos experimentos.

Este modelo se diferencia principalmente de los demás en que tiene en cuenta la forma de la punta y mide la interacción entre los átomos de la muestra, los de la punta y la interacción entre la punta y la muestra (acoplo).

Además se tiene en cuenta el efecto de los procesos de múltiples reflexiones y propagaciones de los electrones entre la punta y la superficie del material.

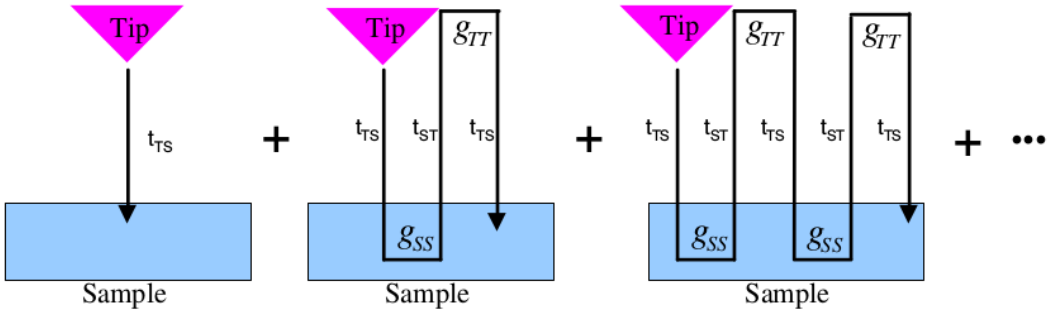

Figura 4: Imagen del funcionamiento de la reflexión múltiple

Como se puede ver en la Figura 4 la intensidad que fluye entre la punta y la muestra por el mismo efecto que la produjo (el efecto túnel) puede volver a subir a la punta y volver a bajar a la muestra indefinidamente (pero cada vez con una probabilidad mucho más baja).

En los métodos más sencillos solo se tiene en cuenta la posibilidad de ir de la punta a la muestra una sola vez por lo que se pierde algo de precisión del experimento.

Con esto y algunas mejoras más se consigue un método más preciso y que se ajusta más a la realidad aunque con un incremento importante en la cantidad de cómputo que hay que realizar.

Con todas estas mejoras la simulación termina teniendo los siguientes pasos:

1. Obtener una descripción de la superficie y la punta por separado. La información de las posiciones de los átomos así como sus características electrónicas son obtenidas mediante métodos basados en DFT a través de unos ficheros (contienen el Hamiltoniano del sistema) que son los que recibe de entrada la simulación STM. Sería algo así como la descripción tridimensional de la superficie, similar a la Figura 5.

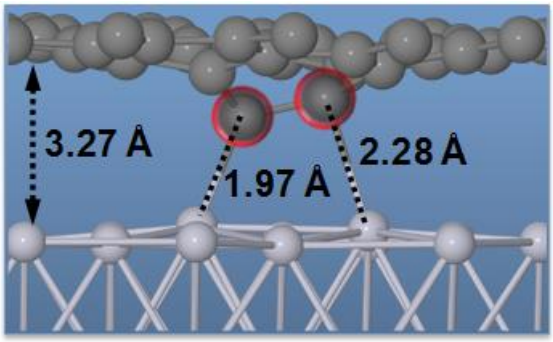

Figura 5: Estructura tridimensional de una muestra

2. A partir de los datos de entrada se calculan las funciones de Green que describen las propiedades electrónicas de punta y muestra.
3. Una vez tenemos calculadas estas funciones de Green lo que se realiza es una simulación del STM, es decir se simula el movimiento de la punta del STM sobre la superficie a distintas distancias y se obtiene la corriente que pasa en cada punto. Con esto ya podemos generar la imagen con la que comparar el experimento real.

Estos pasos que se siguen en las simulaciones no son más que el cálculo de la integral:

$$I = \frac{4\pi e}{h}\int_0^{eV} dE\, Tr[\rho_{ss}(E)D_{ss}^{r}(E)t_{ST}\rho_{TT}(E-eV)D_{TT}^{a}(E-eV)t_{TS}]$$

$D_{TT}^{a}(E-eV)$ y $D_{ss}^{r}(E)$ son calculadas en la función de Green de la forma:

$$D_{TT}^{a}(E-eV) = [1 - t_{TS}g_{SS}^{a}(E)t_{ST}g_{TT}^{a}(E-eV)]^{-1}$$
$$D_{SS}^{r}(E) = [1 - t_{ST}g_{TT}^{r}(E-eV)t_{TS}g_{SS}^{r}(E)]^{-1}$$

El resto de la integral es calculada en la función Scanxy.

Toda la información sobre el método usado para las simulaciones (así como las formulas anteriormente citadas) ha sido obtenida del trabajo de tesis doctoral de José Manuel Blanco Ramos sobre el Estudio teórico del Microscopio de Efecto Túnel con métodos de primeros principios de mayo del 2004 (1).

1.4. Motivación

Debido al método usado y a que hay que tener en cuenta la interacción entre muchas partículas, el tiempo de ejecución de las simulaciones es significativamente alto.

Tipos de simulaciones	Tiempos
Simulaciones pequeñas (<10 átomos y baja precisión)	de 3 horas a 1 día
Simulaciones típicas (más de 10 átomos y precisión media)	de 1 día a 1 semana
Simulaciones ocasionales (más de 50 átomos y/o precisión alta)	1 semana a 1 mes

Tabla 1: Tabla sobre duraciones de las simulaciones

Como se puede apreciar en la Tabla 1 los tiempos de los experimentos suelen ser de días e incluso de semanas.

1.5. Objetivos

La intención final será optimizar y paralelizar el código intentando conseguir el máximo rendimiento posible.

Para ello se empezará con dos funciones (Green y Scanxy) que son las que sospechan los físicos que son las que más tiempo consumen y que a la vez pueden ser paralelizadas.

El código de la primera función (Green) es algo así como:

```
for ikpoint = 1, nkpoints
   assemble_H(datos, ikpoint);
   for ie = 1, nenergy
      grfunc(ie) = ∑ matrices1(ikpoint)
      grret(ie)  = ∑ matrices2(ikpoint)
   end for
end for
```

El código de la segunda función (Scanxy) es de la forma:

```
for x = 1, nx
  for y = 1, ny
    for z = 1, nz
      current0() = ∑ matrices(x, y, z)
    end for
  end for
end for
```

Estas dos funciones están compuestas de una serie de bucles y una serie de suma de matrices lo cual no parece que tenga especial problema para ser paralelizado.

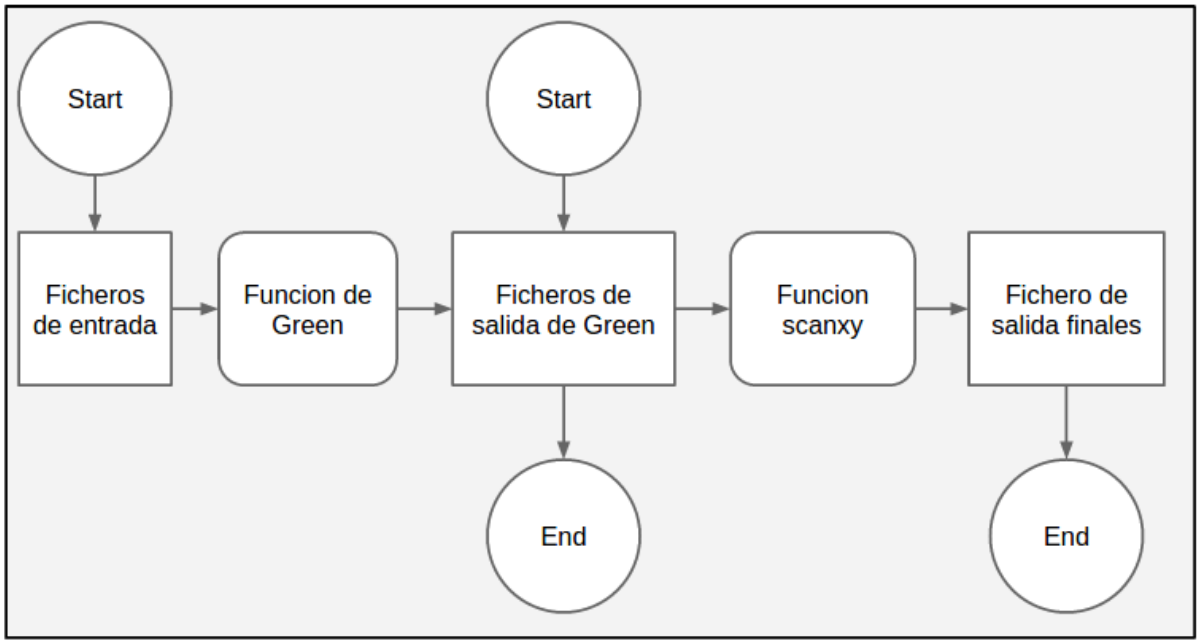

Figura 6: Diagrama del flujo de ejecución del código

En la Figura 6 se muestran los pasos que sigue la ejecución del código. Básicamente con los ficheros de entrada se ejecutara la función de Green y se obtendrán sus resultados que se guardaran en un fichero por si más adelante son necesarios. Después el programa puede terminar o continuar, ejecutando Scanxy y finaliza imprimiendo los ficheros resultantes. Además se podrá empezar usando los ficheros de salida de Green como ficheros de entrada y solo ejecutar la función Scanxy.

1.6. Retos

Además de la propia simulación hemos tenido que tener en cuenta varios retos que tendrá que enfrentar este trabajo para su correcta realización:

1. Usar un software real y con años de uso.
2. El código está escrito en fortran (lenguaje algo anterior a c), lo cual añadirá un plus de complejidad al proyecto ya que no se ha usado en la carrera.
3. El código se compilara con el compilador de Intel el cual es privativo por lo que no se podrá ejecutar en cualquier sitio sino solo en lugares donde esté instalado.
4. Usar MPI en su implementación OpenMPI para paralelizar el código.
5. Ejecutar el programa en el entorno que usan los físicos, con las adaptaciones que haya que realizar.

1.7. Estructura del documento

En este capítulo (Introducción) se han explicado las bases de cómo funciona un STM y como como se simula además de exponer las motivaciones, objetivos y retos de este trabajo.
En el capítulo 2 (Estado del arte) se exponen otros trabajos relacionados con este y que le anteceden
En el capítulo 3(Catálogo de requisitos) se mostraran una serie de conceptos importantes a la vez que se definirán los requisitos del proyecto.
En el capítulo 4 (Metodología) se explicara que metodología de optimización y paralelizado se ha seguido en el desarrollo del proyecto.
En el capítulo 5 (Implementación) se explicara cómo se ha desarrollado el proyecto. Para ello se explicara primero una serie de ideas que se han implementado para facilitar el desarrollo del proyecto y luego se explicara paso a paso como se ha implementado el proyecto en sí.
En el capítulo 6 (Pruebas y resultados) se mostraran las pruebas que se han realizado y los resultados obtenidos a partir de estas.
En el capítulo 7 (Conclusiones y trabajo futuro) se finalizara con las conclusiones y con las mejoras que se podrían implementar en un futuro.

2. Estado del arte

El código usado en este TFG es un código diseñado específicamente para las necesidades del Departamento de Materia Condensada de la UAM. Por ello no existen antecedente de paralelizado de este software.

Existen códigos altamente optimizados y paralelizados como los que calculan la función de densidad (DFT) los que resuelven problemas de dinámica molecular pero estos son usados para otras finalidades distintas.

Tan solo existe un trabajo realizado por Nicolas Boulanger-Lewandowski y Alain Rochefort del 23 de septiembre del 2010 sobre paralelizado de simulaciones de STMs (2).

Para ello, hacen uso de la fórmula:

$$I(r,V) = \frac{4\pi e}{h}\int_{-\infty}^{\infty} \rho_s(E_F^{(s)} + \epsilon)\rho_t(E_F^{(t)} - eV + \epsilon)x\left[f_s\left(E_F^{(s)} + \epsilon\right) - f_t\left(E_F^{(t)} - eV + \epsilon\right)\right]|M_{st}(r)|^2 d\epsilon$$

Con la que paralelizan la ejecución de las simulaciones pero de una forma menos aproximada y más sencilla.

En la Figura 7 se pueden ver algunas simulaciones obtenidas por este método.

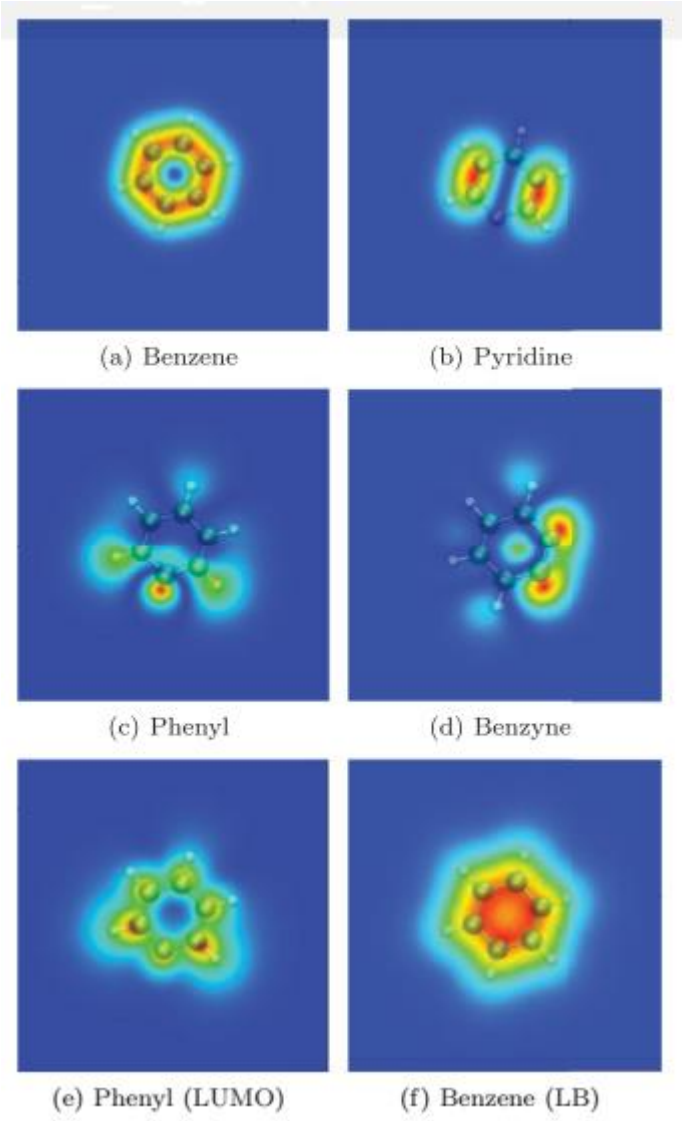

Figura 7: Imágenes de simulaciones obtenidas

3. Catálogo de requisitos

En este capítulo se explicaran primero los actores y sistemas con los que se ha interactuado y luego se describirán los requisitos funcionales y no funcionales del proyecto.

3.1. Actores y sistemas

Para la realización de este trabajo ha sido necesario interactuar con diferentes actores.

Primero tenemos a los Doctores del departamento de Física de la Materia Condensada Pablo Pou Bell (tutor) y Rubén Pérez Pérez que han realizado la función de “clientes”, especificando que es lo que necesitaban y dando las pautas generales de cómo realizarlo.

Luego tenemos al Doctor del departamento de Tecnología Electrónica y de las Telecomunicaciones Iván González Martínez que ha estado ayudando en la parte de la implementación del paralelizado y a la estudiante de doctorado del departamento de Física de la Materia Condensada Lucía Rodrigo Insausti que nos ha brindado su ayuda para que las ideas sobre la implementación pudiesen ser aplicadas sobre el código original que también fue desarrollado en parte por ella.

Finalmente tenemos a los múltiples sistemas de computación masiva que han sido necesarios y que son: el clúster de procesamiento CLS del departamento de física (el más usado), el clúster de procesamiento SPM del mismo departamento de física, el clúster de procesamiento CCC del centro de computación de la UAM, y el clúster de procesamiento Magerit de la Universidad Politécnica de Madrid (UPM).

3.2. Requisitos funcionales

Los requisitos funcionales son los siguientes:

1. El programa permitirá ejecutarlo como si se ejecutase en serie (código original) recibiendo los mismo ficheros de entrada y devolviendo los mismos ficheros de salida.
2. El programa permitirá elegir el número de procesos que se quieren usar y usará los mismos ficheros de entrada que la versión serie y devolverá los mismos ficheros de salida.
3. El código permitirá iniciarse desde el principio con los ficheros de entrada o desde después de la función de Green usando los ficheros de Green como ficheros de entrada.
4. El código permitirá finalizar después de imprimir los ficheros de salida de la funciones de Green o después de imprimir los ficheros de salida finales.
5. El código deberá poder ejecutarse con los mecanismos de arranque existentes actualmente.

3.3. Requisitos no funcionales

Los requisitos no funcionales son los siguientes:

1. El código tendrá que estar escrito en fortran.
2. El código tendrá que poder compilarse con el compilador de Intel para fortran.
3. El código tendrá que usar la implementación OpenMPI de MPI para el paralelizado del código.
4. El código tendrá que poder ejecutarse en el clúster de cómputo CLS del Departamento de la Materia Condensada de la UAM.
5. El código deberá tardar el menor tiempo posible de ejecución, balanceando la carga de trabajo de forma que se minimice el hecho de que haya procesos parados mientras los demás están trabajando.

4. Metodología

Para la metodología se han seguido una serie de pasos que son bastantes estándares para la optimización y paralización del código.

Los pasos que se han seguido son:

1. Estudio del código serie (leyendo el código) e identificación de las partes que se podrían optimizar o paralelizar.
2. Realización de una serie de Profilings (con el que se obtiene el tiempo de ejecución de cada función del programa) con la idea de ver que partes son las de mayor consumo y ver si encaja con los resultados obtenidos en el paso anterior y si esto no pasa, ver cuál es la causa.
3. Optimización del código en la versión serie sobre todo en las partes que luego se van a paralelizar.
4. Paralización del código serie y ver si se distribuye bien la carga de trabajo entre los distintos procesos.
5. Comunicar los procesos entre sí de forma que el resultado de cada proceso pueda ser usado por los demás procesos que lo necesiten.

Para la implementación se han seguido estos pasos (aproximadamente) pero se han ido aplicando de forma cíclica mejorando los tiempos de ejecución poco a poco y optimizando y paralelizando en cada ciclo una parte distinta del código.

5. Implementación

En este capítulo se explicaran los pasos que se han seguido para desarrollar el proyecto. Para ello se explicará primero una serie de elementos que son generales en el proyecto y con los que podrá entender mejor luego los pasos seguidos. Además estos elementos han sido desarrollados a lo largo de todo el proyecto por lo que no tiene tanto sentido ponerlos entre un paso y otro.

Después se explicarán los pasos en sí seguidos para realizar el proyecto. Se han dividido bastante para que se pueda apreciar bien todos los cambios, pruebas y mejoras que se han hecho.

5.1. Versiones del código

Dado que el proyecto se ha realizado con muchos pasos y pensando en hacer muchas mejoras pero pequeñas y dado la dificultad para usar un sistema de control de versiones en el clúster, se ha optado por crear distintas carpetas cada una con una versión distinta del código.

Estas carpetas tienen el nombre mpi seguido de la versión del código en número. En total se han desarrollado 30 versiones distintas. Los nombres de las versiones son: mpi0.0 mpi0.1 mpi0.2 mpi0.3 mpi0.4 mpi0.4.1 mpi0.4.2 mpi0.5 mpi0.5.1 mpi0.5.2 mpi0.5.5t mpi0.5.7t mpi0.5.9t mpi0.6.1x mpi0.6.2x mpi0.6.3x mpi0.6.4x mpi0.7.1r mpi0.7.2r mpi1.0 mpiN0.0 mpiN0.1 mpiN0.1.5 mpiN0.2 mpiN0.3 mpiN1.0 mpiN1.1 mpiN1.2 mpiF2.0 y mpiF2.1.

Algunas de las versiones han sido usadas para medir tiempos de ciertos elementos concretos (mpi*t).

5.2. Scripts de compilación y ejecución

Para poder desarrollar el proyecto de forma eficiente y lo más automáticamente posible se han usado una serie de script escritos en bash.

Los principales son los scripts de compilación (cc.sh) y ejecución (runNtimes.sh).

El script de compilación (cc.sh) es el encargado de compilar las distintas versiones del código. Esté limpia la versión antigua del código y compila el código usando el comando make. Además este script permite elegir una versión concreta a compilar pero también permite compilar todas las versiones de código que haya (para ello compila las carpetas que empiecen por mpi).

El script de ejecución (runNtimes.sh) es el encargado de ejecutar el código en el clúster. Está preparado para poder ejecutar casi todo tipo de pruebas y de formas muy distintas. Para ello se ha ello el código muy modular mediante el uso de múltiples parámetros de entrada. Estos parámetros son:

1. Si se va a ejecutar el código de forma estándar o usando un Profiling.
2. Versión del código a ejecutar
3. Test a ejecutar
4. Número de procesadores a usar

5. Número de veces que se ejecutará el código
6. Si se ejecuta o no el código. Este parámetro está pensado para preparar los archivos de entrada del programa y ejecutarlo a parte, sin el uso de este script.
7. Revisar si los ficheros de salida son correctos (este proceso se explicará en el subcapítulo 5.4 Tests.
8. Qué ficheros son borrados al finalizar el programa
9. Imprimir o no los pasos del script
10. Revisar si otros ficheros de salida son correctos.

Con todos estos parámetros se consigue una alta modularidad y automatización en las ejecuciones consiguiendo por ejemplo: depurar el script, generar los ficheros de entrada necesarios para ejecutar el programa, hacer baterías de pruebas y comprobar los ficheros de salida o no dependiendo de si hace falta.

Para entender esto un poco mejor vamos a explicar cuáles son los pasos que sigue el script una vez se llama.

1. Crea una carpeta donde se guardará todo lo que se use en la ejecución del programa. Esta carpeta está generada de forma que si se llama al script dos veces seguidas cada una use una carpeta distinta y no se pisen. Para ello se hace uso de información del proceso.
2. Se insertan los ficheros de entrada del test elegido y el ejecutable en la carpeta. Es importante que se guarde también el ejecutable ya que si no se podría compilar el ejecutable mientras se usa y podría dar problemas.
3. Se genera, si no existe, la carpeta donde se guardaran los resultados (callgrind o time).
4. Se manda la ejecución del código a la cola de ejecución del clúster. Para ello se usan los ficheros: callgrind.q, time1.q, time2.q, time3.q y time4.q. Mediante estos ficheros el clúster sabe que recursos tiene que reservar, que ejecutable usar y de qué forma.
5. Una vez ejecutado el programa, se comprueban los ficheros de salida (si así lo indican los parámetros).
6. Se guarda el resultado obtenido en su sitio (esto se explicará en el subcapítulo 5.3
7. El paso 4, 5 y 6 se ejecutan tantas veces como número de iteraciones se pidieron. Aun así el paso 5 dado que la salida no cambia de una ejecución a otra (solo debería cambiar el tiempo de ejecución) se ejecuta normalmente una sola vez.
8. Finalmente se borran los ficheros innecesarios.

Con todo esto se ha conseguido ahorrar muchísimo tiempo de implementación y los cambios se han podido realizar más rápidamente.

5.3. Guardado automático de resultados

Como se ha explicado en el apartado anterior de forma rápida los resultados que nos interesan son guardados de forma automática en carpetas específicas. Estos resultados son en el caso de los Profilings un fichero en texto plano que contiene la información del Profiling y en el caso de la ejecución normal el tiempo de ejecución. Cada uno de estos resultados se guarda en carpetas y con nombres, de forma que puedan ser fácilmente identificados. Por ejemplo los tiempos se guardan con el formato: time-(versión código)-(test)-(número de procesos).txt

Luego esos resultados son pasados a un visualizador de Profiling (en el caso de los Profilings) y a una hoja de cálculo (en el caso de los tiempos) donde se podrán estudiar en más detalle.

5.4. Tests

Para poder probar el código se hace uso de diferentes test de “prueba”. Para ello se han usado test de desde 1 segundo hasta 24 horas. En total se han usado 14 tests. Sus nombres son: test1, test2, test2.1, test3, test4, testN1, testN2, testN2.1, testN3, testN4, testFdeno8, testFdeno1024, testFnden8 y testFnden1024.

En cada test se guardan: los ficheros de entrada, los ficheros de salida de la función Green y los ficheros de salida finales. Los ficheros de salida son obtenidos de la versión serie y es contra estos, con los que se compara los ficheros de salida de la ejecución del código.

Además hay que tener en cuenta que estos ficheros de salida pueden llegar a ser de hasta 10GB, algo a tener en ya que no siempre se tiene tanta memoria usable en el disco duro (debido a limitaciones del clúster).

5.5. Diff y Numdiff

Uno de los problemas importantes a los que nos hemos tenido que enfrentar en este proyecto es el hecho de que dado que al paralelizar el código el orden en el que se ejecutan las instrucciones de código no va a ser exactamente el mismo, los resultados obtenido pueden variar levemente.

Un ejemplo sencillo sería ((a+b)+c)+d que uno podría pensar que es lo mismo que (a+b)+(c+d), lo cual es cierto salvo cuando usamos números en coma flotante.

Debido a esto, los ficheros de salida tenían números que variaban en las cifras después de los primeros 10 decimales lo cual no ha sido un problema para la simulación.

Aun así el hecho de comparar los ficheros de salida mediante el comando diff no era posible por lo que se optó por usar el comando numdiff que está preparado para estas situaciones.

5.6. Código serie

Este es el primer paso de la implementación del proyecto. En él, nos hemos familiarizado con el lenguaje usado (fortran) y con el compilador (Intel). Además se ha estudiado (a ojo) el código para intentar ver por donde sería mejor empezar. No se encontró nada especial a parte de las dos funciones (Green y Scanxy) que ya se han explicado antes.

Para el resto de pasos, los primero se centrarán en la función Green para luego pasar a estudiar en los últimos pasos la función Scanxy.

Las medidas de los tiempos de ejecución de los distintos tests con el código serie se pueden ver en el subcapítulo 6.3 Pruebas y resultados de la versión serie

5.7. Profiling

El siguiente paso fue el de realizar diferentes Profilings sobre la versión en serie y sobre diferentes tests pero ejecutando solo hasta cuando termina la función de Green.

Estos Profilings permitieron ver que los tiempos de ejecución de la función Green ocupaban prácticamente todo el tiempo. El resto del tiempo era usado para la lectura de los datos de entrada y la escritura de los resultados, algo que no podía ser paralelizado (simplemente porque no salía rentable y no tenía sentido).

Por tanto con estos Profilings se pudo confirmar que la función importante era Green y que por tanto era la que se debería optimizar y paralelizar.

5.8. Optimización función cexp

Una de las observaciones que se obtuvo de los Profilings era que en un test el tiempo de ejecución de la función cexp (contenida en la función Green) era muy alto. La función cexp es la exponencial compleja la cual no tarda mucho en ejecutarse. El problema es que era llamada aproximadamente 4 millones de veces llegando a ocupar una gran cantidad de tiempo de ejecución.

Por tanto se optimizó haciendo que se llamase un tercio de las veces reescribiendo ligeramente el código, con lo que se redujo significativamente el tiempo que se ejecutaba.

El problema que se observó luego fue que el gran tiempo de ejecución de esta función sólo se produjo en ese test y no se mantuvo para tests más grandes. Por eso al final no fue muy útil y aunque se dejó la mejora, no se han apreciado mejoras en el tiempo significativas.

Todos los datos, Profilings y tiempos obtenidos para este apartado se pueden ver en el subcapítulo 6.4 Pruebas y resultados de la optimización de la función cexp.

5.9. Paralelizado bucle interno de la función Green

Una vez se intentó optimizar el código (con poco éxito) se pasó a empezar a paralelizar el código de la función Green.

Código aproximado de la función Green:

```
for ikpoint = 1, nkpoints
   assemble_H(datos, ikpoint);
   for ie = 1, nenergy
      grfunc(ie) = ∑ matrices1(ikpoint)
      grret(ie)  = ∑ matrices2(ikpoint)
   end for
end for
```

El primer paso en el paralelizado del código fue paralelizar el código del ie que es el bucle interno y por tanto parece el más fácil de paralelizar.

Para ello se distribuyó la carga de la forma más uniforme posible y a cada proceso se le mando datos contiguos (para reducir los fallos de página).Para ello simplemente se divide el número de iteraciones del bucle ie entre el número de procesos y se coge el techo (el entero menor que sea mayor o igual que el número que salga). Ese es el número de iteraciones de cada proceso menos para el último que puede tener menos iteraciones.

Después de esto se pasa al envío y recepción de los datos calculados (matrices). Esta parte que en principio podría haber parecido fácil terminó siendo uno de los problemas más importantes del proyecto.

5.10. Problema de envío y recepción entre procesos

Observando los ficheros de salida se observaba como algunos datos se enviaban bien pero otros no. Y en algunos casos parecía que se enviaban bien la mitad y la otra mitad no.

Después de mucho tiempo probando distintas cosas y distintos tests se pudo separar y encontrar los 2 errores que producían que no se enviasen bien las matrices.

El primero de los errores tiene que ver con cómo fortran maneja los “punteros”. Dado que fortran no tiene punteros (ya existen pero no se usan), este permite otra forma de obtener elementos de matrices o submatrices.

Por ejemplo matriz (1:2,1:2) es una matriz de 2x2 (se empieza a contar en 1 no en 0). Para elegir un elemento de esta solo tenemos que escribir matriz (2,1), con lo que obtendremos el elemento en la posición (2,1) de la matriz.

Por tanto si quiero enviar y recibir una submatriz tendría que ponerlo de la forma matriz (1:2,1) o matriz (1:2,2).

Pero debido a que la implementación en fortran de MPI usa C internamente, se tiene que escribir el código ligeramente distinto.

Para la función de envió se escribe la matriz de la forma matriz (1:2,1) mientras que en la función de recepción se escribe de la forma matriz (1,1) (dando a entender que lo que está indicando es la posición por la que empezar o puntero).

Al final el código de esto termina siendo de la forma:

…
call mpi_recv(GrFunc(1,1,1,1,j)*, …)*
call mpi_recv(gr_ret(1,1,1,1,j)*, …)*
…
call mpi_send(GrFunc(:,:,:,:,j:j)*, …)*
call mpi_send(gr_ret(:,:,:,:,j:j)*, …)*
…

Aun así esto no era suficiente para que funcionase ya que había otro error que se explicara en el siguiente subcapítulo.

5.11. Problema de envío y recepción de la mitad de los datos

Si recuerdan lo que se explicó antes hubo una serie de pruebas en las que salían bien los resultados en la mitad de los casos.

Este error es debido a un parámetro del makefile. El parámetro -r8 es usado en fortran para hacer que los números en coma flotante tengan el doble de tamaño y por tanto se pueda tener más precisión.

El problema es que las funciones de envío y recepción de MPI reciben un parámetro que es el tipo de dato enviado, con el que obtienen el tamaño del tipo de dato. El problema es que este parámetro no cambiaba cuando se usaba r8 y por tanto las funciones de envío y recepción pensaban que tenían que enviar datos del tamaño de la mitad de lo que tenían que enviar realmente.

Para solucionarlo se optó por indicar como tipo de dato a enviar y recibir MPI_BYTE (de un Byte) y como cantidad de datos a enviar se puso *numero elementos*tamaño dato en Bytes*.

En el código en fortran se pasó de:

call mpi_send(gr_ret(:,:,:,:,j:j), ***nElemToSend, MPI_REAL****, ...)*

a:

call mpi_send(gr_ret(:,:,:,:,j:j), ***nElemToSend*sizeof(c), MPI_BYTE****, ...)*

Con estos dos errores corregidos ya se consiguió la primera versión funcional del código paralelizado.

Los resultados numéricos de las pruebas de esta versión del código están en el subcapítulo 6.5 Primeros pruebas y resultados paralelizando (hasta 8 cores)

5.12. Variantes de paralelizado del bucle interno de la función Green

Después de esta primera versión del código en paralelo se optó por intentar enviar los datos (matrices) de distintas formas para ver si el rendimiento mejoraba.

La siguiente versión consistió en enviar los datos de forma desordenada. En la anterior versión primero enviaba el proceso 1 su información al resto, luego el 2, luego el 3, etc…

En esta versión cada proceso enviaba sus datos e iba recibiendo de los demás con forme fuesen llegando.

La siguiente versión consistió en hacer el envió de los datos de forma adelantada. En las versiones anteriores primero cada proceso hacia todos sus cálculos y luego enviaba sus datos. En esta versión cada vez que se calculan una cierta cantidad de datos estos se envían al resto de procesos.

Otra de las mejoras fue la de repartir los cálculos de forma dinámica de forma que cada vez que un proceso procesaba el trozo de los cálculos que tenía, pedía el siguiente trozo y así hasta que se terminan todos los cálculos. Está mejora al final no se implementó ya que se vio que el reparto estático funcionaba muy bien y cada proceso tardaba prácticamente lo mismo.

Una vez tuvimos las nuevas versiones probamos su rendimiento y lo comparamos con el paralelizado inicial. Las potenciales mejoras al final no fueron así y el rendimiento fue el mismo. Según se descubrió mediante Profiling sobre estas versiones, el tiempo de espera hasta poder enviar o recibir y el tiempo de envío o recepción eran en todos los casos mínimos.

Por ello una mejora del envió y la recepción no produjo una mejora del rendimiento y se descartaron estas mejoras.

Los resultados de estos cambios no se muestran ya que no son significativos para el proyecto.

5.13. Problema con el uso de más de un nodo

Hasta este momento las pruebas que se habían hecho sobre el código se habían hecho sobre un nodo.

El clúster principal sobre el que se trabajo fue el clúster CLS del departamento de Física de la Materia Condensada de la UAM. Este clúster consistía de 7 nodos (ordenadores). Cada uno con 2 microprocesadores de 4 núcleos cada uno, sin Hiperthreading.

El problema que había es que cuando se intentaba usar más de 8 cores (más de un nodo), el programa no se ejecutaba.

Gracias a la ayuda sobretodo de Iván González Martínez (ponente), pudimos descubrir que el clúster no estaba preparado para ejecutar MPI de forma correcta.

El error fue remitido al soporte técnico del clúster y en unos días fue corregido. Aun así corrigieron el error pero dejando solo usar hasta 4 nodos (32 cores) por ciertas limitaciones del clúster.

Con esto se pudieron hacer pruebas con hasta 32 cores. Los resultados obtenidos están en el subcapítulo 6.6 Pruebas y resultados paralelización con hasta 32 cores

5.14. Problema de los cores de un mismo nodo

Una vez pudimos probar con hasta 32 cores y se probaron test más largos (de varias horas) empezamos a ver como escalaba la solución (como se reducía el tiempo de ejecución al aumentar el número de cores usados) y que cosas no escalaban bien.

Uno de los problemas más importantes de este trabajo fue relacionado con el escalado.

Cuando se observó este, se apreció como no aumentaba como uno esperaría con hasta 8 cores.

Además esto se repetía de 9 a 16 cores de 17 a 24 cores y de 25 a 32 cores.

Por tanto uno pensaría que algo raro pasaba al usar todos los cores de un nodo por lo que pasamos a hacer un estudio del rendimiento de los cores de un nodo.

Para ello se usaron parámetros especiales de OpenMPI con los que se puede elegir a que nodo y a que core se quiere poner un proceso.

Se probó a medir el tiempo que tardaba el código con cada uno de los cores por separado y luego con cada par de cores. Con estas pruebas se observó un comportamiento atípico del clúster. Básicamente dependiendo que par de cores se usasen, el tiempo variaba y además de forma bastante regular. Se estudió si era problema de acceso a disco duro, tamaño de RAM, velocidad de acceso a RAM, tamaño de cache y velocidad de la cache.

Lo datos obtenidos no fueron concluyentes pero parece que el problema puede ser por cache o por velocidad de acceso a RAM. Para poder ver cuál de estos elementos puede ser el responsable sería necesario un análisis con herramientas específicas con las que se pueda ver donde se satura el sistema. Estas pruebas no se han realizado por falta de tiempo.

Todas las pruebas realizadas para este subcapítulo se pueden ver en el subcapítulo 6.7 Pruebas y resultados sobre el problema de los cores de un mismo nodo.

5.15. Reordenación de iteraciones

Una de las pruebas que se hizo para intentar solucionar el problema anterior y también para ver si se conseguía que se ejecutase el código más rápido fue el reordenamiento de bucles.

Para generar las matrices resultantes de la función Green (que son de 5 dimensiones), estas se generan punto a punto, es decir, primero se calcula un punto, luego el siguiente,…

Por tanto podríamos empezar variando primero la primera dimensión (for i for j for k...) o por la ultima dimensión (... for k for j for i) u en otro orden distinto.

Al probar distintas combinaciones de esto no se obtuvieron cambios en el tiempo de ejecución. Esto parece ser debido al compilador de Intel. El compilador dado que sabe la forma más óptima de calcular matrices y de acceder a ellas puede reordenar los bucles de forma que se haga esto de la forma más rápida posible.

Por tanto estos cambios fueron descartados.

5.16. Paralelizado del bucle externo de la función Green

Una vez se probaron todos los cambios se nos ocurrieron sobre el bucle interior de la función de Green (for ie = 1, nenergy), se pasó a optimizar y paralelizar el bucle exterior (for ikpoint = 1, nkpoints).

Para ello primero se intentó optimizar el cálculo y la memoria de ciertas matrices auxiliares.

Dado que sabíamos cómo íbamos a paralelizar el código y dado que ciertas matrices se generaban varias veces en la versión serie se optó por generarlas en todos los procesos al principio y que luego se usasen las que se necesitaban.

Esto al final no fue demasiado importante ya que el tiempo para generar estas matrices auxiliares no era alto y por tanto se podía despreciar.

Una vez teníamos esto se pasó a paralelizar este bucle.
Los resultados obtenidos con esta mejora pueden verse en el subcapítulo 6.8 Pruebas y resultados sobre el paralelizado del bucle externo de la función Green

5.17. Paralelizado de la función Scanxy

Una vez teníamos paralelizado la función Green y fue testeada con múltiples tests distintos se pasó a optimizar y paralelizar la función Scanxy.

El código aproximado de la función Scanxy es:

```
for x = 1, nx
  for y = 1, ny
    for z = 1, nz
      current0() = ∑ matrices(x, y, z)
    end for
  end for
end for
```

Como se puede apreciar el código es muy parecido al de la función de Green, múltiples bucles que realizan en cada paso sumas y multiplicaciones de matrices.

Por tanto para paralelizar este código se optó por trocear los tres bucles. Para ello se calcula la cantidad de puntos a procesar que es nx*ny*nz y se divide por el número de procesos. Luego se coge el techo de este número y el resultado es el número de puntos a calcular por cada proceso menos el último proceso que puede calcular algunos puntos menos.

Los resultados obtenidos en este apartado se pueden ver en el subcapítulo 6.9 Pruebas y resultados sobre la paralización de la función Scanxy (versión final).

5.18. Ejecución en otros clústeres de procesamiento

Una vez teníamos el código funcionando y se había testeado con múltiples tests de distintos tamaños se pasó a intentar ejecutar el código en otros clústeres de procesamiento.

Esto no solo se hizo con idea de ver el rendimiento en los demás clústeres sino también para ver si los problemas de rendimiento en cada nodo se seguían produciendo en el resto de clústeres o era algo específico del clúster usado.

Al primer clúster al que se paso fue al SPM del departamento de Física de la Materia Condensada de la UAM. Dado que este tenía el mismo sistema de colas y estaba gestionado por los mismos que gestionan el CLS no hubo casi ningún problema para ejecutar el código apropiadamente.

En este clúster se producía también los problemas de escalado en un nodo. En este clúster los nodos eran de 2 procesadores de 6 cores cada uno.
Para ejecutarlo sobre el clúster CCC del centro de computación de la UAM hubo varios problemas. El primero y principal fueron las librerías de compilación. Estas estaban en otros directorios y eran de versiones distintas del compilador de Intel por lo que hubo que cambiar cosas para poder compilar el programa. Además el sistema de colas era distinto por lo que hubo que cambiar el script de ejecución.

Al final no se consiguió ejecutar satisfactoriamente el código ya que daba un error de librerías nada más empezar a ejecutarlo. Este error en el tiempo que se ha tenido no ha podido ser solucionado.

En cuanto al clúster Magerit este ha tenido una serie de problemas. Principalmente fue que no tenían instalado el compilador de Intel ya que es propietario por lo que se tuvo que adaptar el código para que se pudiese compilar con GNU. Esto se explicara en el siguiente apartado.

5.19. Compilación con GNU

Para poder compilar el código con GNU fue necesario cambiar básicamente dos cosas: donde se buscan las librerías y arreglar algunos errores de castings de tipos de datos que GNU no aceptaban.

Con ello se consiguió una versión funcional del código sobre GNU. El problema vino cuando al intentar ejecutar este código sobre Magerit no se enviaba bien la información entre los distintos procesos. Después de esto se realizaron algunas pruebas en otros entornos en los que el código se ejecutaba correctamente sin dar este problema.

La gente del departamento de Física de la Materia Condensada aconsejaron contactar con los del soporte del Magerit para intentar solucionar esto pero debido a restricciones de tiempo esto no se ha podido realizar.

6. Pruebas y resultados

En este capítulo se mostrara las pruebas y los resultados obtenidos.

En cada prueba se obtiene el tiempo que ha tarda en ejecutarse el programa en total.

Es importante tener en cuenta que no se mostraran todos los resultados de todos los test ya que hay algunos resultados que no son significativos.

Además se han realizado muchísimas pruebas. Suponiendo que cada prueba fuese una detrás de otros, 24 horas al día, sin interrupciones y sin ejecutarse 2 pruebas a la vez, tendríamos que se han ejecutado en torno a unas 2000 horas de simulaciones (unos 83 días, casi 3 meses).

6.1. Tests

Para las pruebas se han llegado a usar 14 tests.

Test	test1	test2	test2.1	test 3	test4
k sample	4	64	64	16	64
k tip	4	4	4	4	4
ie sample	5	11	41	41	41
ie tip	5	11	41	41	41

Tabla 2: Tabla tests 1

Test	testN1	testN2	testN2.1	testN3	testN4
k sample	4	64	64	16	64
k tip	4	4	4	4	4
ie sample	4	16	32	64	64
ie tip	4	16	32	64	64
nscanx	6	2	3	6	8
nscany	6	2	3	6	8
nscanz	1	1	1	1	1

Tabla 3: Tabla tests 2

Test	testFnden8	testFdeno8	testFnden1024	testFdeno1024
k sample	4	4	4	4
k tip	4	4	4	4
ie sample	3	3	3	3
ie tip	3	3	3	3
nscanx	2	2	16	16
nscany	2	2	16	16
nscanz	2	2	4	4

Tabla 4: Tabla tests 3

Las tablas 2, 3 y 4 muestran para cada test cuantas iteraciones de los bucles de las funciones Green y Scanxy se realizan. Con esto se puede saber de forma aproximada cuanto va a tardar un test con respecto a otro.

Cada tabla de tests muestra un grupo distinto de tests basados en simulaciones distintas. Dentro de cada tabla los tests se diferencian en la precisión con la que se realizan los cálculos.

Los test de la tablas 2 solo se usan para ejecutar la función de Green por lo que se omiten los valores de Scanxy.

6.2. Conceptos importantes de las pruebas

Para las pruebas realizadas se han realizado estadísticas con la idea de obtener datos significativos y que estén correctamente obtenidos (ya que si no se hace esto se pueden obtener datos que no sean ciertos pero que hayan salido por casualidad).

Para ello se ha tenido en cuenta varias cosas:

1. Cada prueba sobre cada test se realiza al menos 10 veces.
2. En vez de usar la media o la mediana como tiempo de referencia del test se ha tomado el mínimo. Esto es debido a que al medir tiempos de computación los tiempos siempre son muy cercanos y algunas veces algunos valores se desvían pero siempre hacia arriba (tardando más tiempo). Esto es bastante lógico ya que en condiciones iguales y sin ningún programa que choque con el nuestro los tiempos siempre será muy parecido y si no se cumple esto serán mayores pero no menores.
3. También calculamos una “varianza ponderada”, que es la varianza partido el mínimo. Con esto podemos ver rápido si ha pasado algo raro. Si la varianza ponderada es cercana a 0 (menor de 1) los resultados son correctos. Sin embargo si es mayor de 1 entonces algunas pruebas han tardado de más. Dado que usamos el mínimo este problema no es demasiado problemático ya que siempre cogemos el mejor tiempo de todos.
4. Y finalmente calculamos el Speedup, que es el tiempo en serie partido el tiempo en paralelo. Por ejemplo un Speedup de 2 es que la versión en paralelo ha tardado la mitad de la versión en serie.

6.3. Pruebas y resultados de la versión serie

Cada test se ha ejecutado en serie para luego poder calcular el Speedup.

Los tiempos de los tests ejecutando la simulación entera son:

1. Test1: 1 segundo.
2. Test2: 3 horas.
3. Test2.1: 30 minutos.
4. Test 3: 1 hora.
5. Test 4: 4 horas.
6. TestN1: 1 segundo.
7. TestN2: 51 minutos.
8. TestN2.1: 3,7 horas.
9. TestN3: 12,5 horas.
10. TestN4: 25,5 horas.
11. TestFnden8: 1 minuto.
12. TestFnden1024: 8,5 minutos.
13. TestFdeno8: 3,5 minutos.
14. TestFdeno1024: 15 horas.

Es decir tenemos test desde 1 segundo hasta 1 día y muchos con valores intermedios.

Aun así para los resultados se mostraran sobre todo valores de los test que más tardan ya que son los que ofrecen más información sobre el paralelizado de las simulaciones. Con los test de 1 segundo el paralelizado no disminuye el tiempo ya que la mayor parte del tiempo es del arranque del programa.

6.4. Pruebas y resultados de la optimización de la función cexp

Una de las primeras optimizaciones que se hicieron fue sobre la funcione cexp.

Para esta función habíamos encontrado como para un test (test2), el tiempo de ejecución de esta era muy alto.

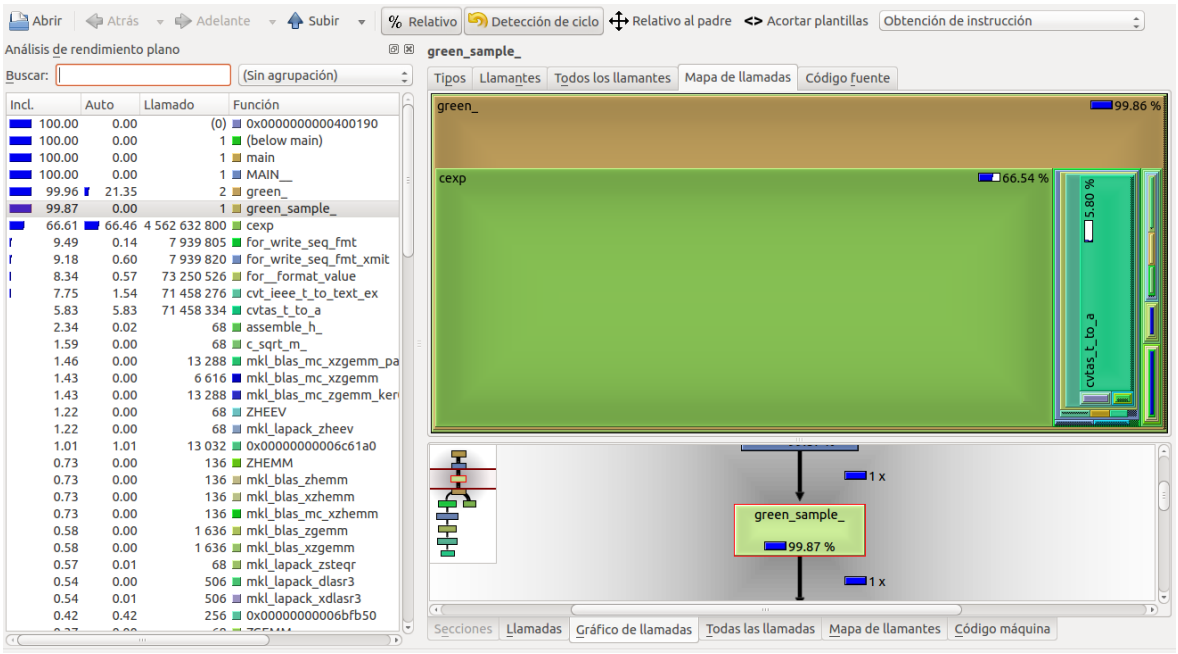

Figura 8: Profiling función Green sobre el test 2

Como podemos ver en la Figura 8 el tiempo de ejecución de la función cexp es el 66% del tiempo de ejecución de la función Green. Esto es debido, como muestra la Figura 8 a que la función cexp se ejecuta en torno a 4,5 mil millones de veces.

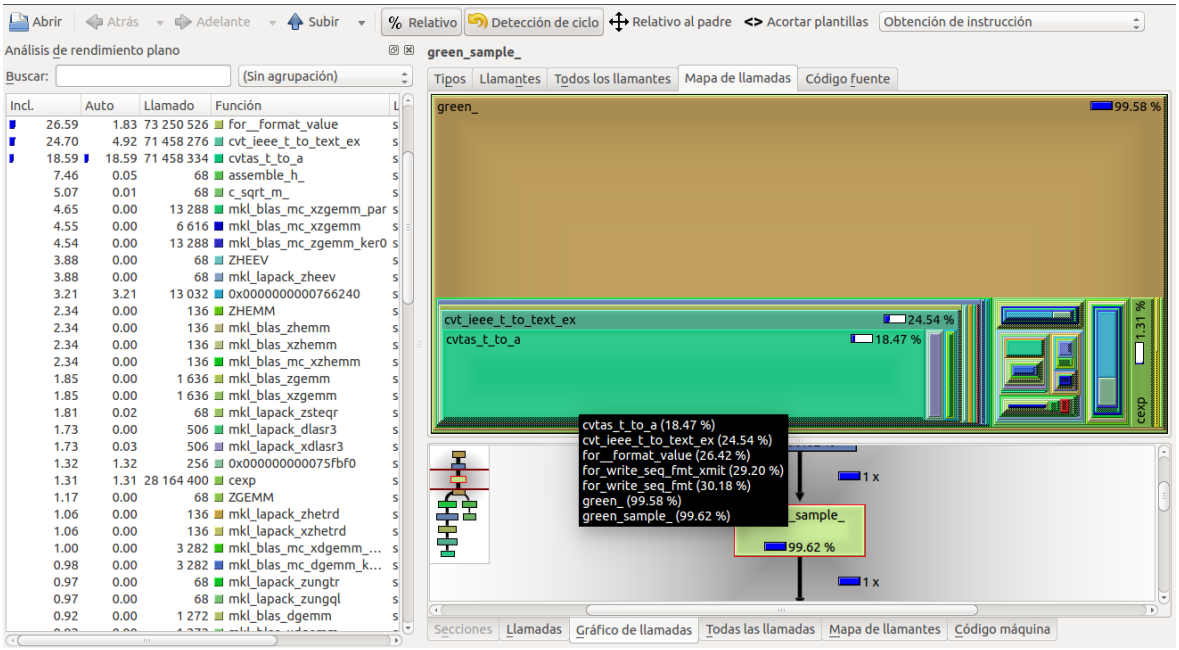

Figura 9: Profiling función Green sobre test 2 después de la optimización de la función cexp

Como se puede apreciar en la Figura 9, después de la optimización la función cexp ya no ocupa casi tiempo de ejecución con un total del solo 28 millones de veces llamada.

Sin embargo para el resto de test (también para los que tardan más que el test 2) esta mejora no se aprecia.

6.5. Primeros pruebas y resultados paralelizando (hasta 8 cores)

Después de optimizar la función cexp se pasó a paralelizar el bucle interno de la función Green.

Mientras estemos mostrando graficas sobre el paralelizado de la función de Green los tiempos solo serán de esta función. Al pasar a la función Scanxy se mostraran graficas con el tiempo y Speedup totales (de todo el programa).

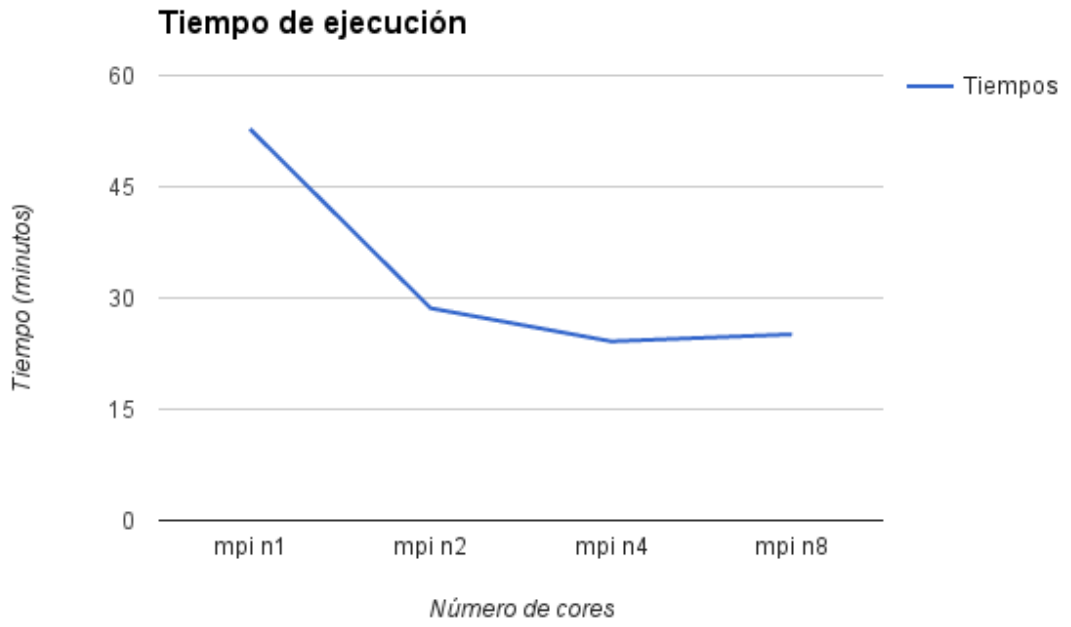

Figura 10: Grafica con el tiempo de ejecución del test 3 variando el número de cores de 1 a 8

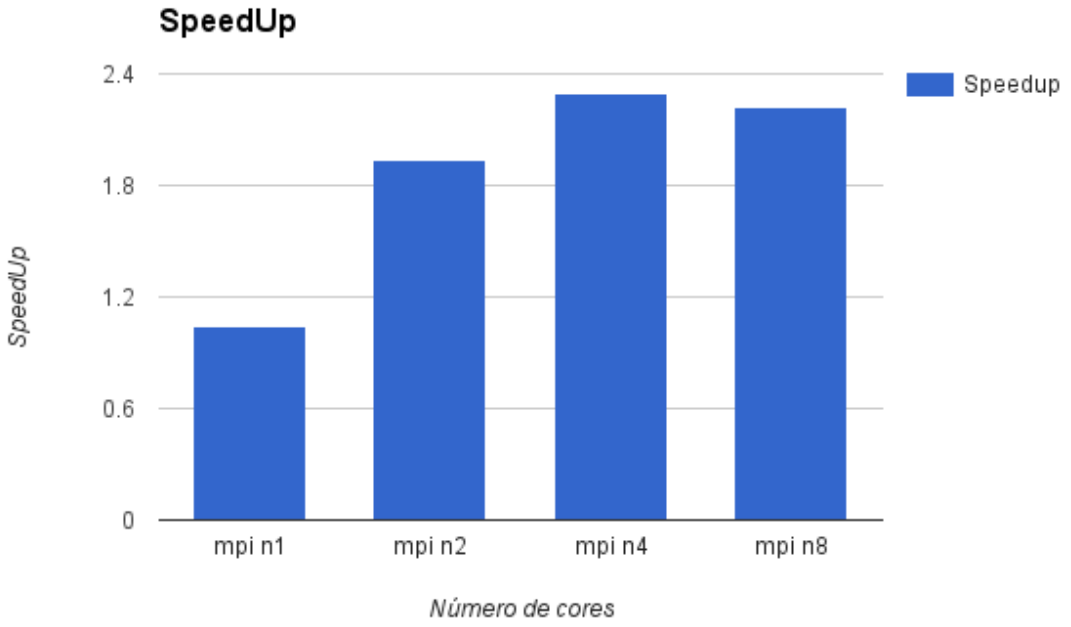

Figura 11: Grafica con el Speedup del test 3 variando el número de cores de 1 a 8

Como se puede apreciar en la Figura 10 y Figura 11 con esta primera paralización conseguimos un Speedup del 2,3 para el test 3 y usando 4 cores no 8 como se explicó en el subcapítulo 5.14 Problema de los cores de un mismo nodo.

En el test 4 se consigue algo más de Speedup, un 2,4 e igual que en el test 3 con 4 cores.

6.6. Pruebas y resultados paralelización con hasta 32 cores

Aquí mostraremos las mismas pruebas sobre la misma versión de código del subcapítulo anterior pero con hasta 32 cores.

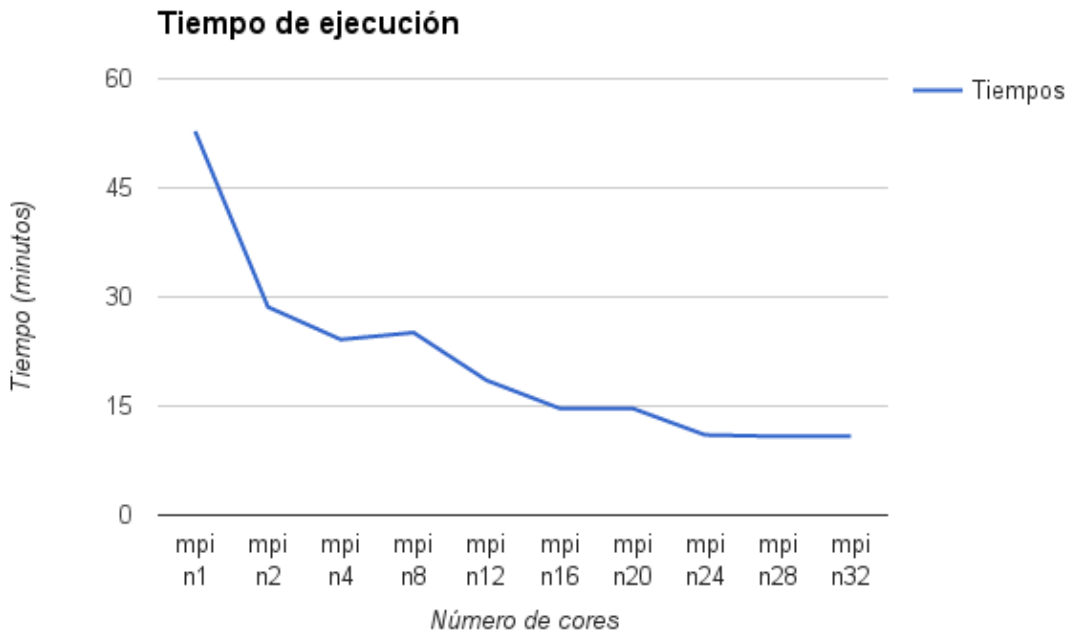

Figura 12: Grafica con el tiempo de ejecución del test 3 variando el número de cores de 1 a 32

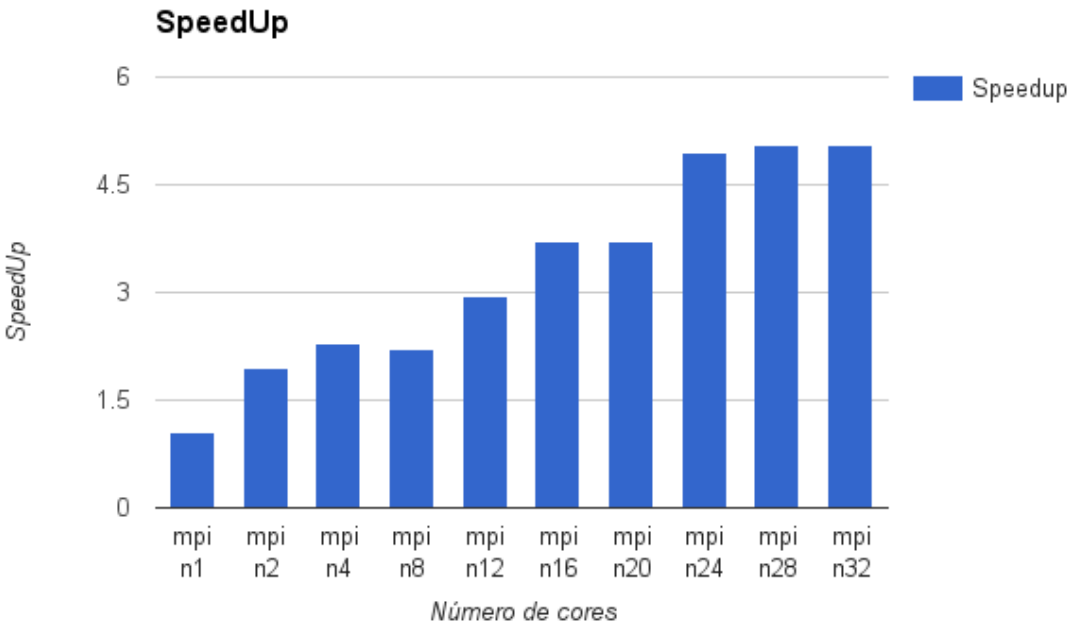

Figura 13: Grafica con el Speedup del test 3 variando el número de cores de 1 a 32

Como se puede apreciar en la Figura 11 y Figura 12 el Speedup aumenta hasta que se satura en 5.

En el test 4 se consigue mejorar un poco llegando a un Speedup de 6 pero con los mismos problemas que en el test 3.

6.7. Pruebas y resultados sobre el problema de los cores de un mismo nodo

Aquí mostraremos los resultados que se han obtenido sobre el problema de los cores.

	0	4	1	5	2	6	3	7
0	X	**133**	118	118	75	75	75	75
4	**133**	X	118	118	75	75	75	75
1	118	118	X	**133**	75	75	75	75
5	118	118	**133**	X	75	75	75	75
2	75	75	75	75	X	**133**	118	118
6	75	75	75	75	**133**	X	118	118
3	75	75	75	75	118	118	X	**133**
7	75	75	75	75	118	118	**133**	X

Tabla 5: Tiempos de ejecución con dos cores en CLS

	0	1	2	3	4	5	6	7	8	9	10	11
0	X	50	50	50	50	50	33	33	33	33	33	33
1	50	X	50	50	50	50	33	33	33	33	33	33
2	50	50	X	50	50	50	33	33	33	33	33	33
3	50	50	50	X	50	50	33	33	33	33	33	33
4	50	50	50	50	X	50	33	33	33	33	33	33
5	50	50	50	50	50	X	33	33	33	33	33	33
6	33	33	33	33	33	33	X	50	50	50	50	50
7	33	33	33	33	33	33	50	X	50	50	50	50
8	33	33	33	33	33	33	50	50	X	50	50	50
9	33	33	33	33	33	33	50	50	50	X	50	50
10	33	33	33	33	33	33	50	50	50	50	X	50
11	33	33	33	33	33	33	50	50	50	50	50	X

Tabla 6: Tiempos de ejecución con dos cores en SPM

La Tabla 5 muestra el tiempo que tarda (en segundos) en ejecutarse un código (una parte de la función de Green) sobre dos cores a la vez. En cada core se pone el mismo código por lo que cada core tardara prácticamente lo mismo que el otro que es el tiempo que se muestra en la tabla.

Como se puede apreciar hay una serie de combinaciones de cores que dan 75 segundos, otras 118 y otras 133. Como se puede apreciar es algo bastante regular por lo que parece que tenga que ver con las caches (que se comparten entre pares de cores y cores en grupos de 4, que son cada microprocesador del nodo).

Uno podría pensar que entonces el tiempo con dos cores debería ser mayor de lo que salía. Esto no pasa ya que al coger el mínimo casi siempre una de las 10 pruebas terminada en pares de cores sin conflictos. Sin embargo este problema si se observa en otra variable de las estadísticas: la varianza ponderada. Esta es máxima casi siempre con 2 cores.

En cuanto a cuando se usan 4 y 8 cores lo que pasa es que la probabilidad de que se pisen dos de ellos es 1 (siempre pasa) por lo que se pierde rendimiento. Aun así con 4 se

consiguen tiempos un poco mejor que con 2 ya que se usan el doble de cores y la penalización es de 1,5. Pero con 8 cores la penalización es mayor que la mejora (frente a 4 cores).

En la Tabla 6 se puede ver el mismo problema solo que sobre el clúster SPM. En este caso hay dos tiempos distintos en vez de 3 pero es problema sigue persistiendo.

Aun así no se ha podido demostrar que el problema es debido a las caches.

6.8. Pruebas y resultados sobre el paralelizado del bucle externo de la función Green

Aquí mostraremos los resultados al paralelizar el bucle externo de la función Green.

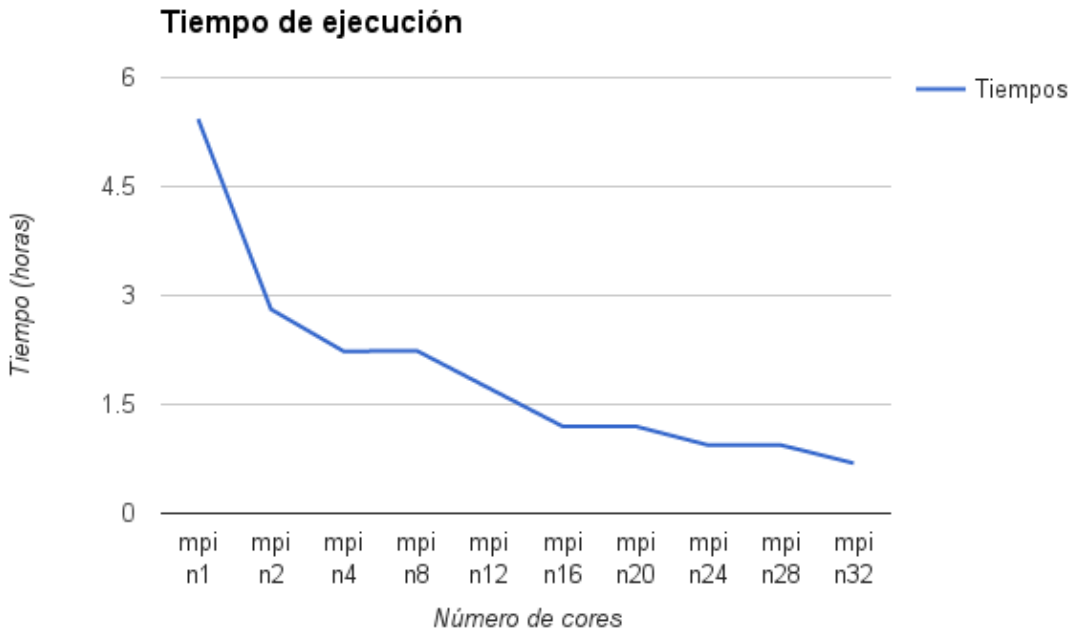

Figura 14: Grafica con el tiempo de ejecución del testN4 variando el número de cores de 1 a 32

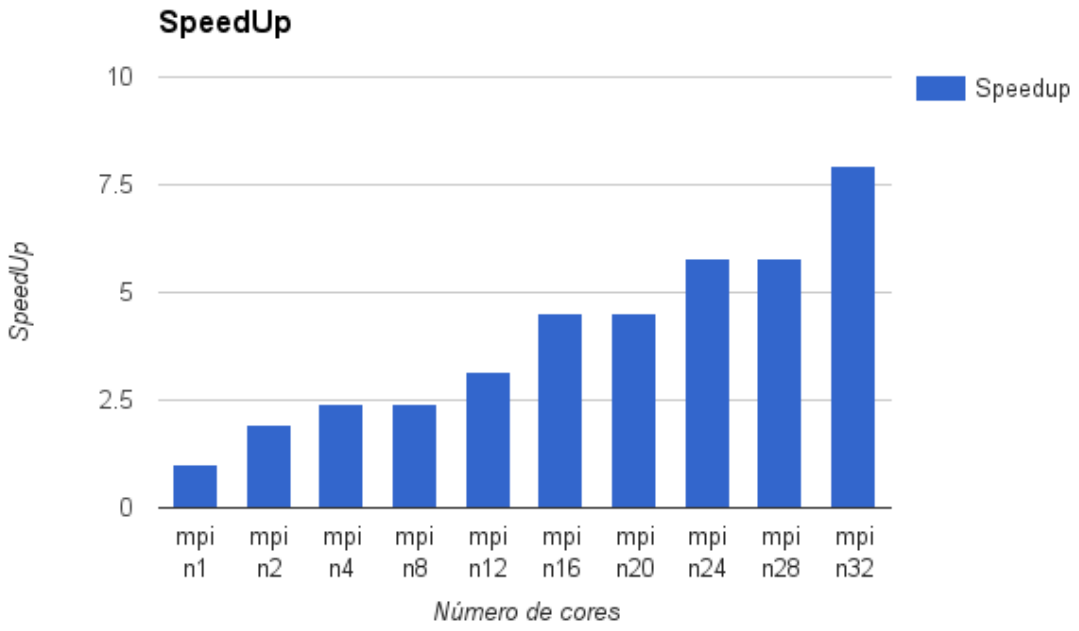

Figura 15: Grafica con el Speedup del testN4 variando el número de cores de 1 a 32

Como se puede apreciar en las Figura 14 y Figura 15 el Speedup mejora. Ahora tenemos un Speedup de casi 8 (7,9).

También hay que tener en cuenta que estos resultados han sido obtenidos sobre el testN4 que es un poco más largo que los anteriores y por tanto puede dar mejores resultados por ello.

6.9. Pruebas y resultados sobre la paralización de la función Scanxy (versión final)

Por ultimo tenemos los resultados finales en los que se ha paralelizado tanto la función de Green como la función Scanxy.

Estos resultados han sido obtenidos mediante el testFdeno1024 que es el que mejor se ajusta a las simulaciones reales que necesitan los físicos hacer.

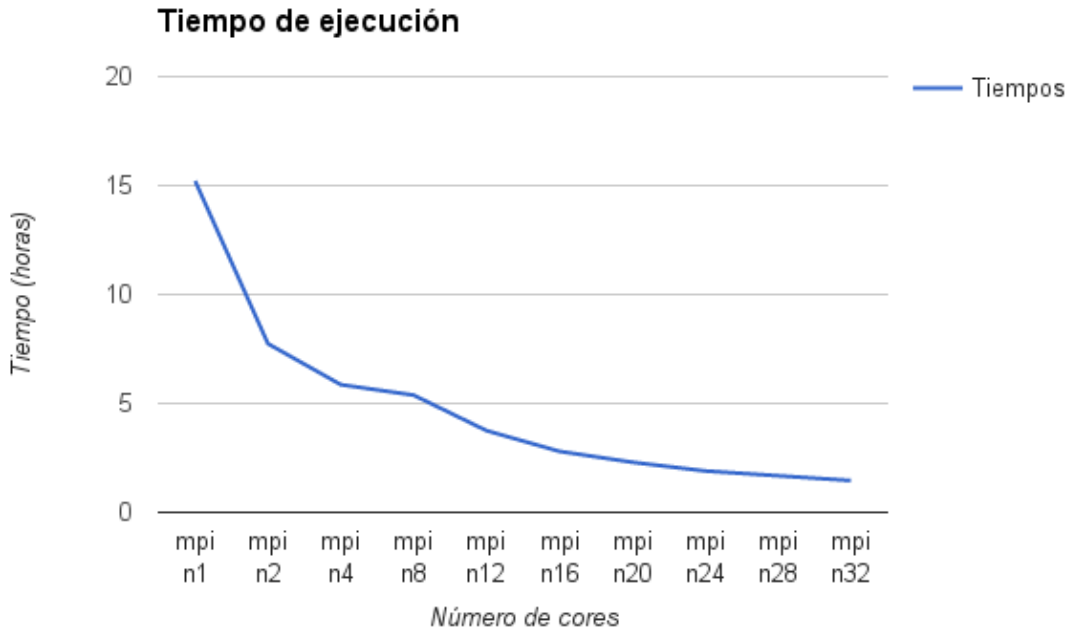

Figura 16: Grafica con el tiempo de ejecución del testFdeno1024 variando el número de cores de 1 a 32

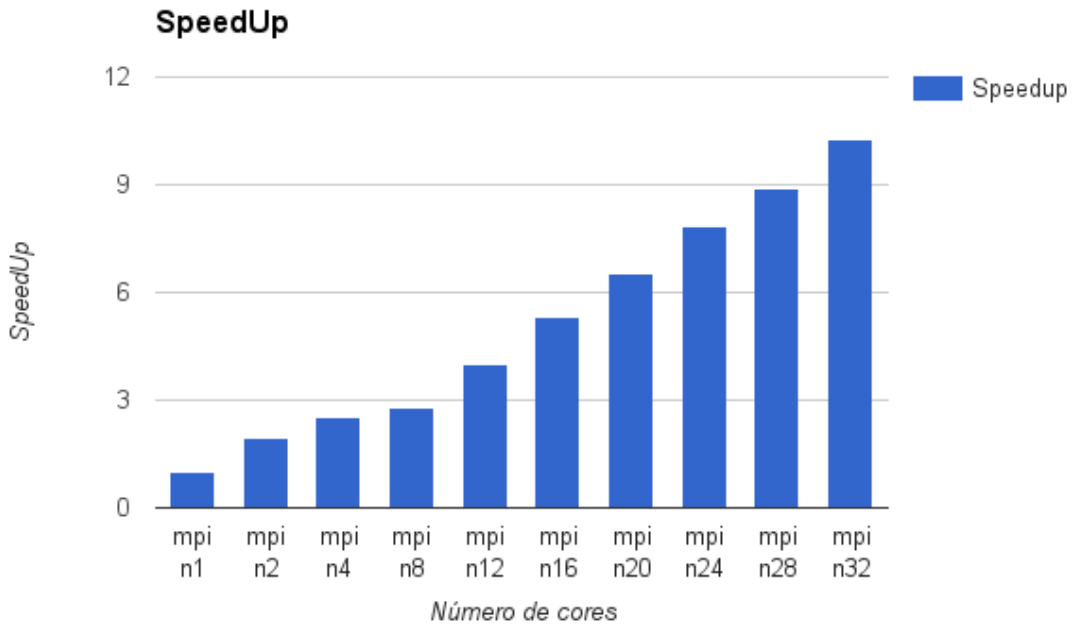

Figura 17: Grafica con el Speedup del testFdeno1024 variando el número de cores de 1 a 32

Como se puede apreciar en la Figura 16 y Figura 17 al final hemos conseguido un Speedup de 10 (un orden de magnitud).

Esto significa que una simulación que tarde una semana ahora tardara 7 horas y una que tarde un mes ahora tardara 3 días.

7. Conclusiones y trabajo futuro

7.1. Conclusiones

Al final aunque partíamos de un código que tiene uso real, escrito en fortran, compilado con Intel y ejecutado en un clúster, hemos conseguido optimizarlo y paralelizarlo reduciendo el tiempo de ejecución en un orden de magnitud (una décima parte del tiempo).

Con esto los físicos podrán generar simulaciones de semanas en horas y de meses en días. Pero además y más importante podrán generar simulaciones que antes eran impensables ya que el tiempo necesario para ello era demasiado alto.

7.2. Trabajo futuro

Como posibles trabajos futuros, proponemos los siguientes:

- Encontrar e identificar por que se produce el problema de los cores en un nodo. Y si puede, solucionarlo.
- Conseguir ejecutar las simulaciones en los clústeres donde no se ha podido: el CCC y Magerit.
- Mejorar la optimización y la paralelizacion que se ha hecho. Esto no parece algo fácil ya que ya se ha conseguido un Speedup de 10 pero los físicos tienen algunas propuestas sobre ello.
- Optimizar y paralelizar otros programas que necesitan los físicos para sus investigaciones.

Referencias

1. **Blanco Ramos, José Manuel.** Estudio teórico del Microscopio de Efecto Túnel con métodos de primeros principios.
2. **Boulanger-Lewandowski, Nicolas y Rochefort, Alain.** Intrusive STM imaging.

Glosario

Microscopio de efecto túnel (STM): microscopio que permite estudiar superficies a escala atómica.

Profiling: proceso por el que se obtiene el tiempo de ejecución y el número de veces que se ha ejecutado cada función. Con esto se puede estudiar que funciones tardan más y cuales menos.

MPI: modelo de programación paralela basada en envió de mensajes. Cada proceso tiene su propia memoria independiente de la de los demás y para intercambiar información estos se mandan la información mediante mensajes.

Clúster de computación: conjunto de ordenadores preparados para trabajar juntos para hacer procesamiento masivo de datos.

Speedup: es una forma de medir el rendimiento de la versión paralela del código frente a la versión serie. Por ejemplo un Speedup de 2 es que la versión en paralelo ha tardado la mitad de la versión en serie.

Anexos

A. Código función Green y Scanxy

Debido a la gran cantidad de fichero del proyecto solo vamos a mostrar el código de las funciones que han sido más importantes, Green y Scanxy.

Código de la función Green:

```
! Program Description
! ========================================================================
! Subroutine to compute the green function and the DOS
!                    CGP summer of 2003
! ========================================================================

! Program Declaration
! ========================================================================

  subroutine green(SmpOrTip,idenomin,einitial,eimag,estep,nenergy,nie,fermi_level, &
                   nkpoints,nk,kpoints,weightk,natoms,natom_ini,natom_end,norb_scan, &
                   norb,norb_ini,norb_end,norbitals,ntypes,itype,ncells, &
                   xcell,xatom,idos,dos,GrFunc,gr_ret,niv_hop,Sdata,xneigh, &
                   nneigh,neighj,nneigh_max,norb_max,neighcell,vecnxn)

       use mpi
       implicit none
       include 'mpif.h'

! Argument Declaration and Description
! ========================================================================
! Input

       character(3), intent(in)                                  :: SmpOrTip
       integer, intent(in)                                       :: idenomin,idos,nenergy,nie
       real, intent(in)                                          :: einitial,eimag,estep,fermi_level
       integer, intent(in)                                       :: nkpoints,nk
       real,dimension(nkpoints,3),intent(inout)                  :: kpoints
       real,dimension(nkpoints),intent(inout)                    :: weightk
       integer, intent(in)                                       :: natoms,natom_ini,natom_end,norb_scan
       integer,dimension(ntypes), intent(in)                     :: norb
       integer, intent(in)                                       :: norb_ini,norb_end,norbitals
       integer, intent(in)                                       :: ntypes
       integer, dimension(natoms), intent(in)                    :: itype
       integer, intent(in)                                       :: ncells
       real, dimension(ncells,3) ,intent(in)                     :: xcell
       real, dimension(natoms,3), intent(in)                     :: xatom
       real, dimension(norb_scan,norb_scan,nenergy), intent(inout)              :: dos
       real, dimension(norb_scan,norb_scan,ncells,ncells,nenergy), intent(inout) :: GrFunc
       complex, dimension(norb_scan,norb_scan,ncells,ncells,nenergy), intent(inout) :: gr_ret
       integer, intent(in)                                       :: nneigh_max,norb_max
```

```
      real, dimension(norb_max,norb_max,0:nneigh_max,natoms), intent (inout)       :: niv_hop
      real, dimension(norb_max,norb_max,0:nneigh_max,natoms), intent (inout)       :: Sdata
      real, dimension(3,1:nneigh_max,natoms), intent (inout)                       :: xneigh
      integer, dimension(natoms), intent (inout)                                   :: nneigh
      integer, dimension(1:nneigh_max,natoms), intent (inout)                      :: neighj
      integer, dimension(3,1:nneigh_max,natoms), intent(inout)                     :: neighcell
      real, dimension(2,3), intent(inout)                                          :: vecnxn

! Output

! Local Variable Declaration and Description
! ============================================================================

    complex, allocatable          :: idn(:,:)         ! identity matrix for H(k) for ideal surface
    complex, allocatable          :: identity(:,:)    ! identity matrix for H(k) for reconstruction
    complex, allocatable          :: green_tot(:,:)   ! green function for the reconstruction (incl.  Selfenergy)
    integer, allocatable          :: iham(:,:)        ! index of hamilt(orb_i,bas_i)
    real, dimension(3)            :: xvect
    real                       :: phase, cexp_value
    integer                       :: norbi,norbj
    integer                       :: in1,in2
    integer                       :: iatom,jatom, ineigh
    integer                       :: iorb,jorb
    integer                       :: icell, jcell
    integer                       :: ibeg, jbeg
    integer                       :: iend, jend
    integer                       :: ibeg2, jbeg2
    integer                       :: iend2, jend2
    integer                       :: i_tipo
    integer                       :: ind
    integer                       :: ie
    integer                       :: ikpoint
    real                       :: aux1, aux2, aux3
    complex                       :: e0
    complex                       :: ener

! LAPACK subroutines
    integer                              :: info
    integer                              :: lwork
    complex, dimension(:),  allocatable     :: work
    integer, dimension(:),  allocatable     :: ipiv


! Procedure
! ============================================================================

  integer :: i,j, kk

  integer :: partk
```

```
  integer :: ikini
  integer :: ikfin
  integer :: ieini
  integer :: iefin
  integer :: nElemToSend

  real :: r
  complex :: c

  real, allocatable :: kkvect(:,:)
  complex, allocatable :: kkhtotal(:,:,:)

  real, allocatable :: dosAux(:,:)
  real, allocatable :: GrFuncAux(:,:,:,:)
  complex, allocatable :: gr_retAux(:,:,:,:)

  integer :: status(MPI_STATUS_SIZE)

  call mpi_comm_size(MPI_COMM_WORLD, sizeMPI, ierror)
  call mpi_comm_rank(MPI_COMM_WORLD, rank, ierror)

! Inicializa
    allocate(iham(natoms,maxval(norb)))  !  JMB 6-7-00
    iham = 0
    ind = 0
    do iatom = 1,natoms
      i_tipo = itype(iatom)
      do iorb = 1,norb(i_tipo)
        ind = ind + 1
        iham(iatom,iorb) = ind
      end do
    end do

! Initialize Lapack variables
    lwork = norbitals*norbitals
    allocate(work(lwork))
    allocate(ipiv(norbitals))

!************************************************************
! ELECTRONIC STRUCTURE CALCULATIONS
!************************************************************

!************************************************************
! Dimension and initialization for hamiltonian and Green Functions
!************************************************************

    allocate(identity(norbitals,norbitals))
    do iorb = 1,norbitals
      do jorb = 1,norbitals
```

```
        if (iorb.eq.jorb) then
          identity(iorb,jorb) = (1.0d0,0.0d0)
        else
          identity(iorb,jorb) = (0.0d0,0.0d0)
        end if
       end do
     end do

     allocate(green_tot(norbitals,norbitals))
     green_tot = (0.0d0, 0.0d0)
     GrFunc = 0.0d0
     !gr_ret = (0.0d0, 0.0d0)
     dos = 0.0d0

!************************************************************
! Calculation of the sample Green function.
!
! We have several loops:
!   ikpoint  over K vectors in reconstrucion G-unit cell
!      ie Energy loop  for calculation of SELFENERGY
!     end ii2 over K vectors in ideal surface G-unit cell
!
!     ie Energy loop  for calculation of sample Greens functions
!       icell,jcell loop over cells for real space projection
!     end ie Energy loop  for calculation of sample Greens functions
!
!   end ikpoint  over K vectors in reconstrucion G-unit cell
!
! What we are doing is..... (formula???)
!
!************************************************************
     e0 = einitial*(1.0d0,0.0d0) + eimag*(0.0d0,1.0d0)
     write(*,'(A11,2f12.6)') ' estep, e0:', estep, einitial

     allocate(kkvect(3,nk))
     allocate(kkhtotal(norbitals,norbitals,nk))

     allocate(dosAux(norb_scan,norb_scan))
     allocate(GrFuncAux(norb_scan,norb_scan,ncells,ncells))
     allocate(gr_retAux(norb_scan,norb_scan,ncells,ncells))

     partk = ceiling(real(nkpoints)/real(nk))
     ikini = modulo(rank,partk)*nk + 1
     ikfin = (modulo(rank,partk)+1)*nk
     ieini = (rank/partk)*nie + 1
     iefin = (rank/partk+1)*nie
     if(ikfin > nkpoints) then
       ikfin = nkpoints
     end if
```

```
if(iefin > nenergy) then
  iefin = nenergy
end if

!Primero calculamos los kvect y los khtotal
do ikpoint = ikini, ikfin
  kkvect(1:3,ikpoint-ikini+1) = kpoints(ikpoint,1:3)

  call assemble_H (kkvect(1:3,ikpoint-ikini+1),iham,kkhtotal(:,:,ikpoint-ikini+1),niv_hop,Sdata,xneigh,nneigh, &
                neighj,norb,norb_max,itype,natoms,norbitals,nneigh_max,ntypes,neighcell,vecnxn)
end do

!Y luego calculamos el resto
do ikpoint = ikini,ikfin
  do ie = ieini, iefin
    ener = e0 + fermi_level*(1.0d0,0.0d0) + (ie-1)*estep*(1.0d0,0.0d0)
    green_tot(1:norbitals,1:norbitals) = ener*identity(1:norbitals,1:norbitals) - &
        kkhtotal(1:norbitals,1:norbitals,ikpoint-ikini+1)

    call inv (green_tot, green_tot, norbitals, norbitals)

    dos(1:norb_scan,1:norb_scan,ie-ieini+1) = dos(1:norb_scan,1:norb_scan,ie-ieini+1) &
      - weightk(ikpoint)/(3.141592)*imag(green_tot(norb_ini:norb_end,norb_ini:norb_end))


    if (idos.ne.1) then
     do icell = 1,ncells            ! Real space projection
       do jcell = 1,ncells
        iend = 0; ibeg =0
        iend2 = norb_ini-1 ; ibeg2 = norb_ini-1
        do iatom = natom_ini, natom_end
         jend = 0 ; jbeg = 0
         jend2 = norb_ini-1 ; jbeg2 = norb_ini-1
         ibeg = iend + 1
         iend = ibeg + norb(itype(iatom)) - 1
         ibeg2 = iend2 + 1
         iend2 = ibeg2 + norb(itype(iatom)) - 1
         do jatom = natom_ini, natom_end
          jbeg = jend + 1
          jend = jbeg + norb(itype(jatom)) - 1
          jbeg2 = jend2 + 1
          jend2 = jbeg2 + norb(itype(jatom)) - 1

          xvect = xcell(icell,1:3) - xcell(jcell,1:3)  +          &
                  xatom(iatom,1:3) - xatom(jatom,1:3)
          phase = dot_product(kkvect(1:3,ikpoint-ikini+1),xvect)
          cexp_value=cexp((0.0d0,1.0d0) * phase)
          GrFunc( ibeg:iend, jbeg:jend, icell, jcell, ie-ieini+1) =           &
               GrFunc(ibeg:iend, jbeg:jend, icell, jcell, ie-ieini+1)       &
```

```
                - weightk(ikpoint)/(3.1416)*imag(green_tot(ibeg2:iend2,jbeg2:jend2) &
                * cexp_value)
            if(idenomin.ne.0) then
              gr_ret(ibeg:iend, jbeg:jend, icell, jcell, ie-ieini+1) =  &
                gr_ret(ibeg:iend, jbeg:jend, icell, jcell, ie-ieini+1) + weightk(ikpoint)   &
                * (real(green_tot(ibeg2:iend2,jbeg2:jend2) * cexp_value)   &
                + (0.0d0,1.0d0)* imag(green_tot(ibeg2:iend2,jbeg2:jend2)  &
                * cexp_value))
             end if
           end do
          end do
         end do
        end do
       end if

     end do     ! End loop ie
   end do       ! End loop ikpoint (main loop)

 nElemToSend = norb_scan*norb_scan*ncells*ncells

 !Recogemos los resultados en el proceso 0
 if (rank == 0) then
   do i = 1, sizeMPI-1
     ikini = modulo(i,partk)*nk + 1
     ikfin = (modulo(i,partk)+1)*nk
     ieini = (i/partk)*nie + 1
     iefin = (i/partk+1)*nie
     if(ikfin > nkpoints) then
       ikfin = nkpoints
     end if
     if(iefin > nenergy) then
       iefin = nenergy
     end if

     do j = ieini, iefin
       call mpi_recv(GrFuncAux(1,1,1,1), nElemToSend*sizeof(r), MPI_BYTE, i, 1, MPI_COMM_WORLD, status,
mpierror)
       if(idenomin.ne.0) then
         call mpi_recv(gr_retAux(1,1,1,1), nElemToSend*sizeof(c), MPI_BYTE, i, 2, MPI_COMM_WORLD, status,
mpierror)
       end if
       call mpi_recv(dosAux(1,1), norb_scan*norb_scan*sizeof(r), MPI_BYTE, i, 3, MPI_COMM_WORLD, status,
mpierror)

       GrFunc(:,:,:,:,j) = GrFunc(:,:,:,:,j) + GrFuncAux(:,:,:,:)
       if(idenomin.ne.0) then
         gr_ret(:,:,:,:,j) = gr_ret(:,:,:,:,j) + gr_retAux(:,:,:,:)
       end if
       dos(:,:,j) = dos(:,:,j) + dosAux(:,:)
```

```
     end do
    end do
  else
    do j = ieini, iefin
     call mpi_send(GrFunc(:,:,:,:,j-ieini+1:j-ieini+1), nElemToSend*sizeof(r), MPI_BYTE, 0, 1, MPI_COMM_WORLD,
mpierror)
     if(idenomin.ne.0) then
      call mpi_send(gr_ret(:,:,:,:,j-ieini+1:j-ieini+1), nElemToSend*sizeof(c), MPI_BYTE, 0, 2, MPI_COMM_WORLD,
mpierror)
     end if
     call mpi_send(dos(:,:,j-ieini+1:j-ieini+1), norb_scan*norb_scan*sizeof(r), MPI_BYTE, 0, 3,
MPI_COMM_WORLD, mpierror)
    end do
  end if

  !Y los enviamos al resto de procesos (para tardar menos se manda al 1 y el 0 pasa a escribir)
  if(sizeMPI > 1) then
    if(rank == 0) then
     call mpi_send(GrFunc(:,:,:,:,:), nElemToSend*nenergy*sizeof(r), MPI_BYTE, 1, 1, MPI_COMM_WORLD,
mpierror)
     if(idenomin.ne.0) then
      call mpi_send(gr_ret(:,:,:,:,:), nElemToSend*nenergy*sizeof(c), MPI_BYTE, 1, 2, MPI_COMM_WORLD,
mpierror)
     end if
     call mpi_send(dos(:,:,:), norb_scan*norb_scan*nenergy*sizeof(r), MPI_BYTE, 1, 3, MPI_COMM_WORLD,
mpierror)
    else if(rank == 1) then
     call mpi_recv(GrFunc(1,1,1,1,1), nElemToSend*nenergy*sizeof(r), MPI_BYTE, 0, 1, MPI_COMM_WORLD,
status, mpierror)
     if(idenomin.ne.0) then
      call mpi_recv(gr_ret(1,1,1,1,1), nElemToSend*nenergy*sizeof(c), MPI_BYTE, 0, 2, MPI_COMM_WORLD,
status, mpierror)
     end if
     call mpi_recv(dos(1,1,1), norb_scan*norb_scan*nenergy*sizeof(r), MPI_BYTE, 0, 3, MPI_COMM_WORLD,
status, mpierror)
     do i = 2, sizeMPI-1
      call mpi_send(GrFunc(:,:,:,:,:), nElemToSend*nenergy*sizeof(r), MPI_BYTE, i, 1, MPI_COMM_WORLD,
mpierror)
      if(idenomin.ne.0) then
       call mpi_send(gr_ret(:,:,:,:,:), nElemToSend*nenergy*sizeof(c), MPI_BYTE, i, 2, MPI_COMM_WORLD,
mpierror)
      end if
      call mpi_send(dos(:,:,:), norb_scan*norb_scan*nenergy*sizeof(r), MPI_BYTE, i, 3, MPI_COMM_WORLD,
mpierror)
     end do
    else
     call mpi_recv(GrFunc(1,1,1,1,1), nElemToSend*nenergy*sizeof(r), MPI_BYTE, 1, 1, MPI_COMM_WORLD,
status, mpierror)
     if(idenomin.ne.0) then
```

```
      call mpi_recv(gr_ret(1,1,1,1,1), nElemToSend*nenergy*sizeof(c), MPI_BYTE, 1, 2, MPI_COMM_WORLD,
status, mpierror)
     end if
     call mpi_recv(dos(1,1,1), norb_scan*norb_scan*nenergy*sizeof(r), MPI_BYTE, 1, 3, MPI_COMM_WORLD,
status, mpierror)
   end if
  end if

! Format Statements
! ========================================================================

122     format(9E15.5)
150     format(2I3,1X,3(3F8.4,2X))

! Free Memory
! ========================================================================
      deallocate(kkvect)
      deallocate(kkhtotal)
      deallocate(dosAux)
      deallocate(GrFuncAux)
      deallocate(gr_retAux)

    deallocate(identity)        ! CGP 01-07-2001
    deallocate(green_tot)       ! CGP 01-07-2001

    deallocate (ipiv)
    deallocate (work)

    return

    end subroutine green_calculation
```

Código de la función Scanxy:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!! Subroutine with all the current calculation stuff             !!!
!!! First of all, we define and allocate some variables for the     !!!
!!! calculation and then starts the scan. 2 bucles that moves the   !!!
!!! tip over one unit cell of the surface (sample)                  !!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    subroutine scanxy(xmin, xmax, ymin, ymax, zmin, zmax, nscanx, nscany, nscanz,   &
                modo_scan, idenomin, ists, icoupling,  &
                nstepx, nstepy, nstepz, zsmpmax, vecnxn_smp, ncells_smp,    &
                norb_scan_smp, natom_smpini, natom_smpend, iwrt_orb)

    use energy
    use tip_def
    use current_def
```

```
   use mpi
   implicit none
   include 'mpif.h'

   real,intent(in)        :: xmin          ! xmin para el scan
   real,intent(in)        :: xmax          ! xmax para el scan
   real,intent(in)        :: ymin          ! ymin para el scan
   real,intent(in)        :: ymax          ! ymax para el scan
   real,intent(in)        :: zmin          ! zmin para el scan
   real,intent(in)        :: zmax          ! zmax para el scan
   integer,intent(inout)   :: nscanx        ! no de pasos en x
   integer,intent(inout)   :: nscany        ! no de pasos en y
   integer,intent(inout)   :: nscanz        ! no de pasos en z
   integer,intent(in)     :: modo_scan
   integer,intent(in)     :: idenomin      ! =0 no denomin, =1 denomin =2 both
   integer,intent(in)     :: ists          ! =1 sts
   integer,intent(in)     :: icoupling     ! =0 old hoppings =1 new hoppings with fireball format
   integer,intent(out)    :: nstepx,nstepy,nstepz  ! auxiliar # of steps
   real,intent(in)        :: zsmpmax
   real,dimension(2,3)    :: vecnxn_smp
   integer, intent(in)    :: ncells_smp
   integer, intent(in)    :: norb_scan_smp
   integer, intent(in)    :: natom_smpini
   integer, intent(in)    :: natom_smpend
   integer, intent(in)    :: iwrt_orb

   real,dimension(2,2)  :: currentaux     ! auxiliar matrix for the conductance
   real,allocatable     :: STS_aux(:,:,:) ! auxiliar matrix for the STS
   real,dimension(3)    :: xtip0, xtip1   ! tip position
   real              :: e1, deltaz
   integer           :: istep, jstep, kstep
   integer           :: index_step
   integer           :: icell
   integer           :: iatom

   integer :: nSteps
   integer :: nStepsPerProcess
   integer :: step
   integer :: j
   real :: r
   real, allocatable :: current0recv(:,:,:,:,:)
   integer :: status(MPI_STATUS_SIZE)

!************************************************************
! SCAN :
!************************************************************

   write(*,*) ''
   write(*,*) '---------------------- SCANNING ----------------------'
```

```
write(*,*) '                Using the formulae: '
write(*,*) '  J=(4*pi*e)/h sum_E(T_12 rho_22 D_r T_21 D_a rho_11)'
write(*,*) '-----------------------------------------------------'
write(*,*) ''

call mpi_comm_size(MPI_COMM_WORLD, sizeMPI, ierror)
call mpi_comm_rank(MPI_COMM_WORLD, rank, ierror)

nSteps = (nscanx+1)*(nscany+1)*(nscanz+1)
nStepsPerProcess = ceiling(real(nSteps)/real(sizeMPI))

! If the info of every atom's current of each cell needs to be written:
if (iwrt_orb.ge.1) then
  open(12, file = 'current_sam_cell.out')
  write(12,*) 'icell  iatom  contribution'
end if

! Inicialize the spatial xyz grid and the matrices of the current (result) and tip position (grid point)
nstepx = nscanx
nstepy = nscany
nstepz = nscanz
if (nscanx.eq.0) nstepx = 1
if (nscany.eq.0) nstepy = 1
if (nscanz.eq.0) nstepz = 1
allocate(current0(0:nscanx,0:nscany,0:nscanz,2,2)) ! LRI COMRPOBAR
allocate(xtip0Ang(0:nscanx,0:nscany,0:nscanz,3))   ! LRI
!allocate(current0(0:nstepx,0:nstepy,0:nstepz,2,2))
!allocate(xtip0Ang(0:nstepx,0:nstepy,0:nstepz,3))
allocate(current0recv(0:nscanx,0:nscany,0:nscanz,2,2))

current0 = 0.d0
current0recv = 0.d0
xtip0Ang = 0.d0
deltaz=(zmax-zmin) !/0.529177249
!if (ists.eq.1) allocate(STS_aux(2,2,nenergy))

step=0
do istep = 0, nscanx
 do jstep = 0, nscany
  do kstep = 0, nscanz
   !write(6,*)'.................................................'
        !write(6,'(a6,i4,a9,i4,a9,i4)')'istep:',istep,'; jstep:', jstep, '; kstep:', kstep

        !*************************************************************
        !               TIP coordinates are set
        !*************************************************************
        !The apex position will be the reference to build the tip
        xtip0 = (/xmin,ymin,zmin/) +     &
        vecnxn_smp(1,1:3)*real(istep)/real(nstepx) + &
```

```
      vecnxn_smp(2,1:3)*real(jstep)/real(nstepy) + (/0.,0.,1./)*deltaz*real(kstep)/real(nstepz)
            xtip1 = xtip0
            ! Tip coordinates translated from the grid point (istep,jstep,kstep) to real coordinates (x,y,z)
            xtip0Ang(istep,jstep,kstep,1:3) = (xtip0(1:3) - (/0.,0.,1./)*zsmpmax)*0.529177249

            !       if (ists.eq.1) then
            !         STS_aux = 0.0d0
            !         call STS(idenomin, modo_scan, icoupling, xtip1, norb_scan_tip, STS_aux, iwrt_orb)
            !         currentSTS(:,:,istep,jstep,kstep,:)=STS_aux(:,:,:)
            !       else
      if(rank*nStepsPerProcess <= step .and. step < (rank+1)*nStepsPerProcess) then
             call currentxy( modo_scan, idenomin, icoupling, zsmpmax, xtip1, xtip0, currentaux, iwrt_orb)
                              current0(istep,jstep,kstep,:,:) = currentaux(:,:)
      endif
            !if (iwrt_orb.ge.1) then
            !  write(12,'(2i5,3f10.4)') istep, jstep, xtip1(1:3)*0.529177249
            !  do icell = 1, ncells_smp
            !    write(12,*) 'cell:',icell
            !    do iatom = natom_smpini,natom_smpend
            !      write(12,'(i5,200E15.5)') iatom, current_sam_cell(icell,hbeg(iatom):hend(iatom))
            !    end do
            !  end do
            !end if
            !end if ! STS or STM

        step = step + 1
      end do   ! kstep scan loop
     end do   ! jstep  scan loop
    end do   ! istep  scan loop

    if (rank == 0) then
     do j = 1, sizeMPI-1
      call mpi_recv(current0recv(0,0,0,1,1), (nscanx+1)*(nscany+1)*(nscanz+1)*2*2*sizeof(r), MPI_BYTE, j, 1,
MPI_COMM_WORLD, status, mpierror)
      current0(:,:,:,:,:) = current0(:,:,:,:,:) + current0recv(:,:,:,:,:)
     end do
    else
      call mpi_send(current0(:,:,:,:,:), (nscanx+1)*(nscany+1)*(nscanz+1)*2*2*sizeof(r), MPI_BYTE, 0, 1,
MPI_COMM_WORLD, mpierror)
    end if

     deallocate(current0recv)

     close(12)

     return

     end subroutine
```